# MUESLI

## *User's Guide*

*Fortran 95 implementation*

Release 2.10.0

Édouard Canot*

Jul 21, 2015

*IRISA/CNRS, Rennes, France

# Contents

**1 Introduction** **4**

**2 What is MUESLI ?** **5**
2.1 MUESLI and other products . . . 6

**3 Getting Started** **7**
3.1 Accessing the `muesli-config` tool . . . 7
3.2 Using `muesli-config` . . . 7
3.3 Compiling and Linking MUESLI programs . . . 8
3.4 Using Makefiles . . . 9
3.5 Setting the environment variables . . . 9
3.6 Using preprocessing . . . 10
3.6.1 General case . . . 10
3.6.2 Compiler dependant functions . . . 10
3.7 Command line argument . . . 10

**4 FML (Numerical Library)** **11**
4.1 First seeds of MUESLI . . . 11
4.1.1 The fundamental derived type: `mfArray` . . . 11
4.1.2 Creating and Setting `mfArrays` . . . 11
4.1.3 Displaying `mfArrays` . . . 12
4.1.4 Parameters and Global Variables . . . 14
4.1.5 Information returned at run-time by MUESLI . . . 15
4.1.6 Programmer's facilities in MUESLI . . . 16
4.2 Working with `mfArrays` . . . 16
4.2.1 Setting and Getting elements in an `mfArray` . . . 16
4.2.2 Concatenating and Replicating `mfArrays` . . . 20
4.2.3 Doing Operations with `mfArrays` . . . 21
4.2.4 Boolean `mfArrays` . . . 24
4.2.5 Doing Linear Algebra . . . 27
4.2.6 Data Analysis with `mfArrays` . . . 30
4.2.7 Other numerical tasks . . . 35
4.2.8 Saving/reading `mfArrays` to/from disk . . . 38
4.2.9 Sparse `mfArrays` . . . 39
4.3 Advanced topics . . . 47
4.3.1 Matrix Properties . . . 47
4.3.2 Physical Units in `mfArrays` . . . 48
4.3.3 Performance issues . . . 52
4.3.4 Managing Memory . . . 53
4.3.5 Creating your own `mf-` or `ms-` routines . . . 53
4.3.6 Numerical precision . . . 54
4.3.7 Debugging in MUESLI . . . 54
4.3.8 Exchanging data with MATLAB . . . 62
4.4 Equivalence between MUESLI routines, MATLAB commands and f90 intrinsic functions . . . 63
4.4.1 from MUESLI routines . . . 63
4.4.2 from MATLAB commands . . . 64
4.4.3 from f90 intrinsic functions . . . 65

**5 FGL (Graphical Library)** **66**
5.1 Creating and Numbering Figures . . . 66
5.2 High-level Plotting Routines . . . 66
5.3 Interactive Routines . . . 79
5.4 Annotating a Figure . . . 81
5.5 Text, Symbols and Fonts . . . 84
5.5.1 About the character sets used . . . 84
5.5.2 Greek alphabet and other symbols . . . 84
5.5.3 Fonts available . . . 86
5.6 Printing Figures . . . 88

5.7 Low-level Graphic Object's Manipulation . . . . . 89

**Index** **92**

# 1 Introduction

This document describes the MUESLI Fortran 95 library.

MUESLI is freely available at the following address: http://people.irisa.fr/Edouard.Canot/muesli

More information can be found in the following documents:

- *MUESLI Installation Guide*
- *MUESLI Reference Manual*
- *MUESLI Inside*

The source code of MUESLI comes with many examples which can also give some help.

Copyright © 2003-2015, Édouard Canot, IRISA/CNRS, Rennes, France.

Bugs reports or comments: mailto:Edouard.Canot@irisa.fr

Thanks to the MUESLI users: Samih Zein, Rabé Badé, Mohamad Muhieddine, Eduardo Mendes, Flàvio Aquino, Saulo Benchimol Bastos, Mohamed Lamine Sahari, Merline Flore Djouwe, Salwa Mansour, Paweł Biernat, Jerzy Wróbel, Antti Kemppinen, Corentin Beaucé.

### Typographic convention

In this guide, coloring is used to differentiate source code and execution output. Lines from source code will be always displayed in this color, whereas input/output in a terminal will be always displayed in this color.

MUESLI routine names displayed in this color are included in the final index.

For sake of clarity, array constructors use brackets '`[`' and '`]`' (Fortran 2003) instead of '`(/`' and '`/)`' (Fortran 95).

### About the name

*Muesli:* loose mixture of mainly rolled oats and often also wheat flakes, together with various pieces of dried fruit, nuts, and seeds. There are many varieties, some of which also contain honey powder, spices, or chocolate. (from http://en.wikipedia.org)

### Credits

*Cover photograph:* the photo on the cover is copyrighted by Niderlander. It can be used under a Limited Royalty Free License (see http://www.dreamstime.com/muesli-and-candied-fruits-imagefree4006423).

# 2 What is MUESLI ?

The MUESLI library is splitted in two parts, which correspond to the following Fortran modules:

- FML (Fortran Muesli Library) contains all necessary materials to numerically work with a dynamic array (dynamic in size, type and structure), called `mfArray`.
  To work with FML, your Fortran 95 source code must include the statement:

  ```
  use fml
  ```

- FGL (Fortran Graphics Library) contains graphics routines which use the `mfArray` objects.
  To work with FGL, your Fortran 95 source code must include the statements:

  ```
  use fml
  use fgl
  ```

Nearly all of MUESLI routine names begin either by the two characters `"ms"` (for Muesli Subroutine), or by `"mf"` (for Muesli Function).

## 2.1 MUESLI and other products

Actually, MUESLI is trying to be a near-clone of MATFOR®, especially of its philosophy. Note that most of routine names are derived from the ones of MATLAB. It's approach is also close to the MATRAN's one but, in our humble opinion, MUESLI is easier to use.

### MATLAB®

Matlab is an interactive software for algorithm development, data visualization, data analysis, and numeric computation. Add-on toolboxes extend the Matlab environment to solve particular classes of problems. Its syntax (vector- and matrix-oriented) makes it a powerfull high-level language.
Matlab is a commercial product: http://www.mathworks.com/products/matlab

### OCTAVE

Octave is a high-level language, primarily intended for numerical computations. It provides a convenient command line interface for solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with Matlab.
Octave is a free software: http://www.octave.org

### SCILAB

Scilab is a scientific software for numerical computations providing a powerful open computing environment for engineering and scientific applications. Scilab includes hundreds of mathematical functions with the possibility to add interactively programs from various languages. It has sophisticated data structures, an interpreter and a high-level programming language.
Scilab is a free software: http://www.scilab.org

### MATRAN

Matran is a Fortran 95 wrapper that implements matrix operations and computes matrix decompositions using Lapack and the Blas. It provides the flavor and convenience of coding in matrix oriented systems like Matlab, Octave, etc. By using routines from Lapack and the Blas, Matran allows the user to obtain the computational benefits of these packages with minimal fuss and bother.
Matran is a free software: http://www.cs.umd.edu/~stewart/matran/Matran.html

### MATFOR®

Matfor is a set of Fortran 95 libraries that enhances your program with dynamic visualization capabilities, shortens your numerical codes and speeds up your development process. By adding a few lines of Matfor codes to your Fortran program, you can easily visualize your computing results and perform real-time animations.
Matfor is a commercial product: http://www.ancad.com/overview.html

# 3 Getting Started

The aim of this section is to verify that MUESLI is properly installed and works well.

## 3.1 Accessing the `muesli-config` tool

First of all, you must verify that the MUESLI configuration tool works on your system. In a terminal, type the following command:

```
$ muesli-config
```

You should obtain something like:

```
MUESLI configuration tool
MUESLI is a numerical and graphical library with a Fortran 95 API.
Copyright (C) 2003-2012, Édouard CANOT, IRISA/CNRS, Rennes, France.

Usage: muesli-config [OPTION]

Known values for OPTION are:

  --prefix            MUESLI installation prefix
  --f90compiler       f90 compiler name
  --f90flags          pre-processor and compiler flags
  --libs              library linking information
  --libs-static       static link
  --help              displays this help and exits
  --version           library information
  --all               all information (library + compiler)
```

If you obtain instead:

```
bash: muesli-config: command not found
```

then, you should add the location of the `muesli-config` script in your `PATH` environment variable, for example (for bash shells):

```
$ export PATH=<path to muesli-config>:$PATH
```

If you are not sure that MUESLI has been installed on your system, you can also install it anywhere inside your own home directory. See the *MUESLI Installation Guide* for doing this.

## 3.2 Using `muesli-config`

As you can see when you invoke it, `muesli-config` has many options giving some information about the MUESLI configuration. Only one option must be used at a time, but one of them (`--all`) displays all known information:

```
$ muesli-config --all
```

returns for example:

```
library source path:
/local/ecanot/tools/fortran/f90-modules/MUESLI/muesli/GNU_GFC

library installation path:
/local/ecanot/lib/muesli

library documentation path:
/local/ecanot/share/muesli

library release:
2.5.2

build date:
Mon Jan 30 16:15:53 CEST 2012

compiler version:
GNU Fortran (GCC) 4.5.2
Copyright (C) 2010 Free Software Foundation, Inc.

make options:
make CFG=release MEMTR=no TCOV=no PTRTR=no PROF=no
```

All this information is important in the case where you find a bug or an incorrect behavior of the MUESLI library.

Next subsection shows you how to take advantage of the `muesli-config` script to easily compile and link your programs.

## 3.3 Compiling and Linking MUESLI programs

Three options of the `muesli-config` script are useful: `--f90compiler`, `--f90flags` and `--libs`.

The first one, `--f90compiler`, gives the name of the Fortran 90 compiler used to build the installed version of MUESLI. For compatibility reasons with the precompiled modules of MUESLI, the same compiler (and even sometimes the same version) must be used to generate MUESLI programs. For example:

```
$ muesli-config --f90compiler
gfortran-4.5
```

The second one, `--f90flags`, is related to the compilation options you should use. It returns pathes for module's inclusion or other specific flags. For example:

```
$ muesli-config --f90flags
-D_GNU_GFC -I/local/ecanot/lib/muesli -fno-range-check -w -O2 -funroll-loops -fo
mit-frame-pointer -fopenmp
```

A simple way to include the returned strings in the command line is to use backquotes:

```
$ ‘muesli-config --f90compiler‘ ‘muesli-config --f90flags‘ -c myprog.f90
```

this will produce the object file `myprog.o`. It is strictly equivalent to:

```
$ gfortran-4.5 -D_GNU_GFC  -I/local/ecanot/lib/muesli -fno-range-check -w -O2 -f
unroll-loops -fomit-frame-pointer -fopenmp -c myprog.f90
```

The last option, `--libs`, returns all the required linking flags and libraries. For example:

```
$ muesli-config --libs
-fopenmp -Wl,--rpath,/local/ecanot/lib/muesli /local/ecanot/lib/muesli/libfgl.so
 /local/ecanot/lib/muesli/libfml.so /local/ecanot/lib/muesli/libmfplot.so -lstdc
++ -Wl,--rpath,/local/ecanot/lib/lapack/GNU_GFC /local/ecanot/lib/lapack/GNU_GFC
/libblas.so -Wl,--rpath,/local/ecanot/lib/lapack/GNU_GFC /local/ecanot/lib/lapac
```

```
k/GNU_GFC/liblapack.so -Wl,--rpath,/local/ecanot/lib/lapack/GNU_GFC /local/ecano
t/lib/lapack/GNU_GFC/libblas.so -L/usr/lib -lX11 -lpng -lz -lpthread -lreadline
-lhistory /local/ecanot/lib/muesli/dummy_papi.o -Wl,--rpath,/opt/gcc-4.5/lib/opt
/gcc-4.5/lib/libgfortran.so.3
```

Therefore, to build your executable `myprog`, you should use the following shorter command:

```
$ `muesli-config --f90compiler` -o myprog myprog.o `muesli-config --libs`
```

## 3.4 Using Makefiles

Using a Makefile is an easy way to produce your executables. This means that you will avoid to repeatidly type appropriate commands in the terminal.

For example, in order to build the `myprog` executable from the `myprog.f90` source file, you should copy the following command lines in a file named `Makefile` (actually a GNU-Makefile):

```
myprog : myprog.o
        `muesli-config --f90compiler` -o $@ $< `muesli-config --libs`

.SUFFIXES :
.SUFFIXES : .f90 .o

%.o : %.f90
        `muesli-config --f90compiler` `muesli-config --f90flags` $(CFLAGS) -c $<

clean :
        rm -f *.o myprog
```

(don't forget to insert the <TAB> character instead the eight spaces at the beginning of some lines).

Then, type the `make` command to process the Makefile.

## 3.5 Setting the environment variables

In order to have a ready-to-use library, three different environment variables must be set correctly: `PATH`, `LD_LIBRARY_PATH` and `MFPLOT_DIR`.

The `PATH` environment variable contains a list of directories containing executable binaries (*e. g.* , the compiler name, the `muesli-config` script, etc). Type:

```
$ echo $PATH
```

to verify.

If the installation of MUESLI is in the RELEASE mode (actually, it is certainly the case), you will have to include also in the `LD_LIBRARY_PATH` variable the path of the HDF5 shared library.

If, at execution, the following error message is displayed:

```
$ myprog
./myprog: error while loading shared libraries: libhdf5.so: cannot open shared
object file: No such file or directory
```

you must register the path of `libhdf5.so` by setting:

```
$ export LD_LIBRARY_PATH=<path to libhdf5.so>:$LD_LIBRARY_PATH
```

In the case where you obtain problems with other shared libraries, you have to build a static executable by typing:

```
$ `muesli-config --f90compiler` -o myprog myprog.o `muesli-config --libs-static`
```

After that, if you obtain again an error (only when using FGL) like:

```
$ myprog
  ...
%MFPLOT, Unable to read font file: grfont.dat
%MFPLOT, Use environment variable MFPLOT_FONT to specify the location of the MFPLOT
         grfont.dat file.
MFPLOT /xw: Couldn't find program "mfxwin_server" in the directory named
MFPLOT /xw: in your MFPLOT_DIR environment variable, or in any directory
MFPLOT /xw: listed in your PATH environment variable.
  ...
```

then you must set the following variable[1]:

```
$ export MFPLOT_DIR=<MUESLI install dir>
```

## 3.6 Using preprocessing

### 3.6.1 General case

For many reasons, you may need to use preprocessing in your program. You can do it via, as usual, the sharp character symbol (#) in the first column of your source files.

Additionally, you can introduce the definition of any preprocessing symbol in your Makefile by using the universal `'-D'` flag. This must be added in the automatic rule command which transform the source file (`%.f90`) into object file (`%.o`).

In any case, don't forget that your source file must end with the '.F90' extension, and not '.f90'.

### 3.6.2 Compiler dependant functions

In the case where you have to use some compiler dependant functions, be aware that the following preprocessing symbols are already defined, only one for each compiler used:

`_GNU_GFC`

`_INTEL_IFC`

## 3.7 Command line argument

The two following usual forms are possible:

- the `'iargc()'` and `'getarg()'` F77 compatibility functions [2];
- the `command_argument_count()` and `get_command_argument()` F2003 functions.

[1] For a permanent setting, you should insert the appropriate command in your '.bash_profile' (resp. '.cshrc') if you are using 'bash' (resp. 'csh' or 'tcsh').

[2] declare 'iargc' as an integer, without the 'external' attribute; this is the recommended rule valid for all the compilers supported in MUESLI.

# 4 FML (Numerical Library)

**Typographic note**

Knowing that Fortran is not a case sensitive programming language, we use lower and upper case names only for the sake of clarity.

## 4.1 First seeds of MUESLI

### 4.1.1 The fundamental derived type: `mfArray`

An `mfArray` object is an automatic and dynamic 2D array. This means that at run time, its shape, type and structure can change automatically without the need to do it explicitly. The program deduces them from the context.

The `mfArray A` is first declared as follows:

```
type(mfArray) :: A
```

We must see now how to initialize it and how to work with it.

In all examples presented in this document, except when it is specified, all the variables are of type `mfArray`.

### 4.1.2 Creating and Setting `mfArrays`

An `mfArray` can be assigned a scalar:

```
A = 7
```

a vector:

```
A = [ 1, 2, 3, 4, 5 ]
```

or a matrix:

```
A = reshape( [ 1, 2, 3, 4, 5, 6 ], [ 2, 3 ] )
```

which has 2 rows and 3 columns. The array which is contained in A has exactly the same shape than the Fortran array in the *RHS* of the previous assignement. Moreover, the elements are stored in the same order; after all, it is just a special Fortran array.

Many functions exist to help you in creating classical and well known mfArrays. A random array is a typical example:

```
A = mfRand( 2, 3 )
```

Even if your `mfArray A` is a scalar or a vector, it will be stored internally as a rank-2 array; therefore you must specify its two dimensions. So, for example, to create a vector of 5 random values, it will writes:

```
A = mfRand( 1, 5 )
```

for a row vector, and

```
A = mfRand( 5, 1 )
```

for a column vector.

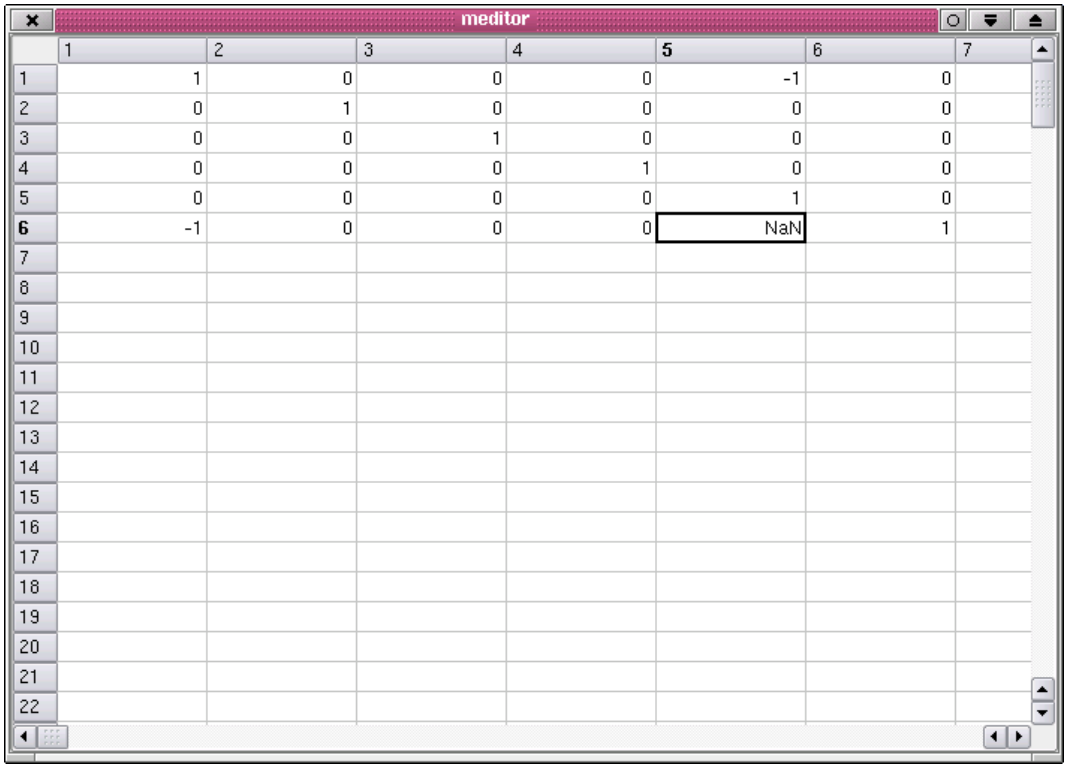
Other MUESLI routines are available to build identity matrices (`mfEye`), zeros matrices (`mfZeros`), ones matrices (`mfOnes`), equally spaced vectors (`mfLinSpace`), etc. In the *MUESLI Reference Manual*, they are listed into the subsection named "Elementary Matrix Manipulation Functions".

Note also that the `mf()` routine converts many Fortran objects into `mfArrays`. The use of this latter routine is sometimes useful when you must put an `mfArray` argument to a routine.

Finally, a graphical tool should also be mentioned: 'meditor' is a small Qt-based application[3], like a spreadsheet, which can be launched by the `msMedit` routine. This tool could be helpful to interactive programs.

▶ Example #4-1:

```
A = mfEye( 6 ) ! Identity matrix of order 6
call msMedit( A ) ! interactive modification (3 elements modified)
call msDisplay( A, "A" )
```




▷ Output of ex. #4-1:

```
A =

   1.0000    0.0000    0.0000    0.0000   -1.0000    0.0000
   0.0000    1.0000    0.0000    0.0000    0.0000    0.0000
   0.0000    0.0000    1.0000    0.0000    0.0000    0.0000
   0.0000    0.0000    0.0000    1.0000    0.0000    0.0000
   0.0000    0.0000    0.0000    0.0000    1.0000    0.0000
  -1.0000    0.0000    0.0000    0.0000       NaN    1.0000
```

### 4.1.3 Displaying `mfArrays`

For many reasons, calling the `msDisplay` subroutine is the only way to print `mfArrays` on the screen. Indeed, the mfArray derived type contains private components hence only a part of the whole structure is relevant to print. Moreover, printing a big array on the screen must be done by a specific routine, because information should be clearly presented.

Anyway, a try to directly print an `mfArray`, as *e. g.* :

[3] perhaps this application is not available on your system.

```
A = mfRand( 2, 3 )
print *, "A = ", A
```

will ordinarily lead to an error at compile time (INTEL ifort compiler):

```
print *, "A = ", A
-----------------^
fortcom: Error: myprog.f90, line 13: This derived-type object in an
input-output list cannot have private components.   [A]
```

msDisplay is a sophisticated routine which takes care of, among other things, the number of columns of your terminal [4]. It prints the array in a pretty form which is column-oriented, like in MATLAB.

Here are some examples of **mfArrays** displayed.

▶ Example #4-2:

```
A = mfRand( 2, 3 )
call msDisplay( A, "A" ) ! default : only 5 digits are printed
```

▷ Output of ex. #4-2:

```
A =

   0.9839    0.2753    0.8098
   0.7000    0.6611    0.9100
```

▶ Example #4-3:

```
call msFormat("long") ! changes the default : 15 digits are printed
A = mfRandN(5,5) ! generates random numbers with normal distribution
call msDisplay(A, "A") ! fits the number of cols to the terminal width
```

▷ Output of ex. #4-3:

```
A =

Columns 1 through 4

 -1.25110306696529   0.89704366553922  -1.92831962464287  -0.15523126577206
  2.17898954410140   1.57784530974119   0.20224358652870  -0.70089607528083
  1.36288491055927   0.80171278372511   0.72856226839328   0.36313288326467
 -0.67409792738829   2.29746275397409   0.53498742614739   0.03939436198377
 -0.32474732088845  -1.88330423996211  -0.94996868780511  -0.67354050025227

Column 5

  0.91939656133688
 -0.15673634460146
  0.71741927783678
 -0.07448835251978
  2.31970783552094
```

[4] if the current display doesn't fit your terminal, your can force the number of columns by setting manually the environment variable **MF_COLUMNS**. See also the description of the **msSetTermCol()** routine in the *MUESLI Reference Manual*

▶ Example #4-4:

```
call msFormat("short") ! returns to the default behavior
A = mfRandN(5,1)*1.0e+5
call msDisplay(A, "A") ! factorizes a common exponent
```

▷ Output of ex. #4-4:

```
A =

  1.0E+05 *

  -1.2511
   0.8970
  -1.9283
  -0.1552
   0.9194
```

Two interesting options available in msDisplay are `head` and `tail`. They show only few elements of the `mfArray`, avoiding a print of a large number of elements.

▶ Example #4-5:

```
A = mfLinSpace(0.0d0,1.0d0,1001) ! long row vector
A = .t. A                        ! long col vector
call msDisplay(A, "A", head=5, tail=5)
```

▷ Output of ex. #4-5:

```
A =

   0.0000
   0.0010
   0.0020
   0.0030
   0.0040
 ... (991 lines skipped)
   0.9960
   0.9970
   0.9980
   0.9990
   1.0000
```

### 4.1.4 Parameters and Global Variables

A few number of parameters and constants are predefined in MUESLI, in order to allow the user to quickly (or easily) set mfArrays to certain values.

These parameters include mathematical usual real contants (MF_PI $\approx 3.14159$, MF_E $\approx 2.71828$) as well as special IEEE value (MF_INF for infinity and MF_NAN for Not-a-Number).

The complex constant MF_I (pure imaginary complex number) is also useful to build complex `mfArrays`.

▶ Example #4-6:

```
A = .t. mf( [ 1, 2, 3, 4, 5 ] ) * MF_I
call msDisplay(A,"A")
```

▷ Output of ex. #4-6:

```
A =

  0.0000 + 1.0000i
  0.0000 + 2.0000i
  0.0000 + 3.0000i
  0.0000 + 4.0000i
  0.0000 + 5.0000i
```

Section 1 in the *MUESLI Reference Manual* describes other real parameters. On the other hand, only three (special) `mfArray` parameters are defined in MUESLI: `MF_EMPTY`, `MF_COLON` and `MF_END`.

At the user level, it is not possible to declare other `mfArray` parameters, because the `mfArray` derived type contains private components. However, the `msSetAsParameter` allow a protection of the data; this routine must be used explicitly, once the `mfArray` object is initialized.

▶ Example #4-7:

```
permeability = 1.0d-12                                  ! initialization
call msSetAsParameter( permeability, param=.true. ) ! protecting the mfArray

permeability = 0                                        ! should rise an error
```

▷ Output of ex. #4-7:

```
(MUESLI assignment(=):) ERROR: cannot change LHS !
                       it is a protected array (pseudo-parameter).

traceback requested from MUESLI :
At line 209 of file mod_mfdebug.F90
Traceback: (Innermost first)
...
Called from line 14 of file muesli_test.f90
```

Lastly, `STDIN`, `STDOUT` and `STDERR` are global variables which can be modified by the user.

### 4.1.5 Information returned at run-time by MUESLI

As opposed to a classical work with standard Fortran 90 variables, when working with the MUESLI library, the compiler doesn't know the actual type of an `mfArray` because it is a dynamic object.

So, the major part of the errors you obtain will arise at run-time and not at compile-time. When a MUESLI routine encounters an error (or even an unexpected situation), it always prints a message which should be self-explanatory for the user.

MUESLI messages are of three kinds: *info* (just give some information), *warning* (expect some strange results) or *error* (something wrong has happened). MUESLI rarely stops the program execution, except for internal errors (bugs). However, for *error* kind messages, MUESLI waits for an answer from the keyboard to resume.

The default behavior can be changed by the user, at any location in his program, via the `msSetMsgLevel` subroutine (see the *MUESLI Reference Manual*). In particular, the quiet mode is useful when using a batch system: in case of error, the program stops without waiting for a user answer.

### 4.1.6 Programmer's facilities in MUESLI

When your program executes a long task, it can be useful to see the progress report. In the case where a costly part is located inside a loop, the `msPrintHashes` routine displays a line of hashes, progressively during the loop execution. Another routine, `msPrintProgress`, is more sophisticated since it prints the percentage of the work done and an estimation of the remaining time.

In other circumstances, when the program prompts the user for the name of a file or whatever, the user will find useful to use the `mfReadline` routine. This routine and others (`msReadHistoryFile`, `msWriteHistoryFile`) brings to the user the ability to naviguate (via the keyboard arrows) in an history and to edit the line.

## 4.2 Working with `mfArrays`

### 4.2.1 Setting and Getting elements in an `mfArray`

Two routines are used to set and get elements of an `mfArray`: `msSet` and `mfGet`. Both can be used with a single element, an array section, or even whole row(s) and whole column(s). The position of the elements is given as two lists of integers which are the row and column indexes.

▶ Example #4-8:

```
A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay(A,"A")
call msSet(0.5d0,A,2,2) ! changes one element
call msDisplay(A,"msSet(0.5d0,A,2,2), A")
```

▷ Output of ex. #4-8:

```
A =

    1     4     7
    2     5     8
    3     6     9

msSet(0.5d0,A,2,2), A =

   1.0000    4.0000    7.0000
   2.0000    0.5000    8.0000
   3.0000    6.0000    9.0000
```

▶ Example #4-9:

```
A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay(A,"A")
call msSet(0.25d0,A,[2,3],[2,3]) ! changes an array section
call msDisplay(A,"msSet(0.25d0,A,[2,3],[2,3]), A")
```

▷ Output of ex. #4-9:

```
A =

    1     4     7
    2     5     8
    3     6     9

msSet(0.25d0,A,[2,3],[2,3]), A =

   1.0000    4.0000    7.0000
   2.0000    0.2500    0.2500
   3.0000    0.2500    0.2500
```

► Example #4-10:

```
  A = reshape( [ (i+0.5, i = 1, 9) ], [ 3, 3 ] )
  call msDisplay(A,"A")
  call msSet(0.0d0,A,MF_COLON,[3]) ! changes a whole column
  call msDisplay(A,"msSet(0.0d0,A,MF_COLON,[3]), A")
```

▷ Output of ex. #4-10:

```
A =

   1.5000    4.5000    7.5000
   2.5000    5.5000    8.5000
   3.5000    6.5000    9.5000

msSet(0.0d0,A,MF_COLON,[3]), A =

   1.5000    4.5000    0.0000
   2.5000    5.5000    0.0000
   3.5000    6.5000    0.0000
```

► Example #4-11:

```
  A = reshape( [ (i+0.5, i = 1, 9) ], [ 3, 3 ] )
  call msDisplay(A,"A")
  call msSet(MF_EMPTY,A,[2],MF_COLON) ! suppresses a whole row
  call msDisplay(A,"msSet(MF_EMPTY,A,[2],MF_COLON), A")
```

▷ Output of ex. #4-11:

```
A =

   1.5000    4.5000    7.5000
   2.5000    5.5000    8.5000
   3.5000    6.5000    9.5000

msSet(MF_EMPTY,A,[2],MF_COLON), A =

   1.5000    4.5000    7.5000
   3.5000    6.5000    9.5000
```

▶ Example #4-12:

```
A = [ (i, i = 1, 12) ]
call msDisplay(A,"A")
call msDisplay(mfGet(A,3),"mfGet(A,3)") ! gets one element in a vector
```

▷ Output of ex. #4-12:

```
A =

    1     2     3     4     5     6     7     8     9    10    11    12

mfGet(A,3) =

    3
```

▶ Example #4-13:

```
A = reshape( [ (i, i = 1, 25) ], [ 5, 5 ] )
call msDisplay(A,"A")
! gets an array section (non contiguous)
! 'i .to. j .step. k' creates a integer sequence
call msDisplay(mfGet(A,MF_COLON, 1 .to. 5 .step. 2 ),                &
               "mfGet(A,MF_COLON, 1 .to. 5 .step. 2 )")
```

▷ Output of ex. #4-13:

```
A =

    1     6    11    16    21
    2     7    12    17    22
    3     8    13    18    23
    4     9    14    19    24
    5    10    15    20    25

mfGet(A,MF_COLON, 1 .to. 5 .step. 2 ) =

    1    11    21
    2    12    22
    3    13    23
    4    14    24
    5    15    25
```

Under certain circumstances, you may find that the routine `msPointer` could be more useful (see the examples below). However, you must know exactly the current data type of the `mfArray`. Moreover, it locks some internal properties about the matrix, therefore you must not forget to call `msFreePointer` instead of a simple nullify of the pointer; this latter routine remove the pointer link but also unlocks the internal properties.

▶ Example #4-14:

```
! real(kind=MF_DOUBLE), pointer :: f90_ptr(:,:)

A = [ 1, 2, 3 ]
call msDisplay(A,"A")
call msPointer(A,f90_ptr) ! special use of: f90_ptr => A
f90_ptr(1,2) = 777
print *, "'msPointer(A,f90_ptr); f90_ptr(1,2) = 777'"
call msDisplay(A,"A")
call msFreePointer(A,f90_ptr) ! implies: f90_ptr => null()
```

▷ Output of ex. #4-14:

```
A =

    1     2     3

'msPointer(A,f90_ptr); f90_ptr(1,2) = 777'

A =

    1   777     3
```

▶ Example #4-15:

```
! real(kind=MF_DOUBLE), pointer :: f90_ptr_vec(:)

A = mfEye(3)
call msDisplay(A,"A")
call msPointer(A,f90_ptr_vec) ! special use of: f90_ptr_vec => A
f90_ptr_vec(2) = 7
print *, "'msPointer(A,f90_ptr_vec); f90_ptr_vec(2) = 7'"
call msDisplay(f90_ptr_vec,"f90_ptr_vec")
call msFreePointer(A,f90_ptr_vec) ! implies: f90_ptr_vec => null()
```

▷ Output of ex. #4-15:

```
A =

    1     0     0
    0     1     0
    0     0     1

'msPointer(A,f90_ptr_vec); f90_ptr_vec(2) = 7'

f90_ptr_vec =

    1     7     0     0     1     0     0     0     1
```

Lastly, `msSet` accepts out of range indexes for element(s) to be modified: the matrix shape is modified accordingly, with zero filling.

▶ Example #4-16:

```
A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay(A,"A")
call msSet(0.75d0,A,4,5) ! out of range indexes => reshapes the matrix
call msDisplay(A,"msSet(0.75d0,A,4,5), A")
```

▷ Output of ex. #4-16:

```
A =

    1     4     7
    2     5     8
    3     6     9

msSet(0.75d0,A,4,5), A =

   1.0000    4.0000    7.0000    0.0000    0.0000
   2.0000    5.0000    8.0000    0.0000    0.0000
   3.0000    6.0000    9.0000    0.0000    0.0000
   0.0000    0.0000    0.0000    0.0000    0.7500
```

### 4.2.2 Concatenating and Replicating `mfArrays`

`mfArrays` concatenation is obtained via two operators (`.hc.` and `.vc.`) which place two matrices side by side, either horizontally or vertically.

▶ Example #4-17:

```
A = transpose( reshape( [ (-i, i = 1, 6) ], [ 2, 3 ] ) )
call msDisplay(A,"A")
B = reshape( [ (i, i = 1, 15) ], [ 3, 5 ] )
call msDisplay(B,"B")
z = A .hc. B ! A or B could be empty
call msDisplay(z,"A .hc. B")
```

▷ Output of ex. #4-17:

```
A =

   -1    -2
   -3    -4
   -5    -6

B =

    1     4     7    10    13
    2     5     8    11    14
    3     6     9    12    15

A .hc. B =

   -1    -2     1     4     7    10    13
   -3    -4     2     5     8    11    14
   -5    -6     3     6     9    12    15
```

▶ Example #4-18:

```
A = mfLinSpace( 1.0d0, 10.0d0, 10 ) ! equally spaced vector
A = .t. mfReshape( A, 5, 2 ) ! '.t.' operator takes the transpose
call msDisplay(A,"A")
A = A .vc. [ 11.0d0, 12.0d0, 13.d0, 14.d0, 15.d0 ]
call msDisplay(A,"A .vc. [ 11.0d0, 12.0d0, 13.d0, 14.d0, 15.d0 ]")
```

▷ Output of ex. #4-18:

```
A =

    1     2     3     4     5
    6     7     8     9    10

A .vc. [ 11.0d0, 12.0d0, 13.d0, 14.d0, 15.d0 ] =

    1     2     3     4     5
    6     7     8     9    10
   11    12    13    14    15
```

Another way to construct a bigger matrix is to use `mfRepMat` which applies concatenation of the same tile-matrix in both direction.

▶ Example #4-19:

```
A = reshape( [ (i, i = 1, 4) ], [ 2, 2 ] ) ! this is the tile
call msDisplay(A,"A",mfRepmat(A,2,4),"mfRepmat(A,2,4)")
```

▷ Output of ex. #4-19:

```
A =

    1     3
    2     4

mfRepmat(A,2,4) =

    1     3     1     3     1     3     1     3
    2     4     2     4     2     4     2     4
    1     3     1     3     1     3     1     3
    2     4     2     4     2     4     2     4
```

### 4.2.3 Doing Operations with `mfArrays`

Usual arithmetic operation $(+, -, *, /)$ can be applied to **`mfArrays`**, because the corresponding Fortran intrinsic operators have been overloaded by specific routines. More generally, nearly all the operations and functions available in Fortran 90 can be applied to **`mfArrays`** (however sometimes under a different command name, see the section 4.4).

The general rule could be the following one: try first to use the function you want (of course, prefixed by `"mf"`). If you obtain an error during the compilation of your program, then see the *MUESLI Reference Manual* which describes all available MUESLI routines. If your function doesn't exist, you can in a last resort implement it by reading the guidelines in the section 4.3.5.

Most of the mathematical functions work element-wise. Using a Fortran 90 specific term, MUESLI functions are *elemental*. This is the case for the multiplication (`*`), the exponential function (`mfExp`), the square-root function (`mfSqrt`), the sine function (`mfSin`), etc.

MUESLI contains a number of specialized mathematical functions: `mfErf` for the Error Function, `mfGamma` for the Γ function; it also contains routines about Bessel functions (`mfBesselI`, `mfBesselJ`, `mfBesselK`, `mfBesselY`) and Complete Elliptic Integrals of first and second kinds (`msEllipKE`).

We cannot show an example of each MUESLI function. In the MUESLI source package, there is a folder named *tests* which contains many program tests. Please refer to these source files to understand how the routines work. Only few examples are presented here.

▶ Example #4-20:

```
A = mfOnes(3) - mfEye(3)*MF_I ! mixing real and complex kinds
call msDisplay(A,"A")
```

▷ Output of ex. #4-20:

```
A =

  1.0000 - 1.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
  1.0000 + 0.0000i   1.0000 - 1.0000i   1.0000 + 0.0000i
  1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 - 1.0000i
```

▶ Example #4-21:

```
A = mfMagic(3)
call msDisplay(A,"A")
A = A * A ! element-wise operation
call msDisplay(A,"A * A")
```

▷ Output of ex. #4-21:

```
A =

    8     1     6
    3     5     7
    4     9     2

A * A =

   64     1    36
    9    25    49
   16    81     4
```

▶ Example #4-22:

```
A = mfOnes(2,3)*3.0d0
call msDisplay(A,"A")
call msDisplay(A/2.0d0,"A/2.0d0" )
```

▷ Output of ex. #4-22:

```
A =

    3     3     3
    3     3     3

A/2.0d0 =

   1.5000    1.5000    1.5000
   1.5000    1.5000    1.5000
```

▶ Example #4-23:

```
  x = mfLinSpace( -1.5d0, 1.5d0, 4 )
  call msDisplay(x,"x",mfASin(x),"mfASin(x)") ! result will be complex
```

▷ Output of ex. #4-23:

```
x =

  -1.5000   -0.5000    0.5000    1.5000

mfASin(x) =

 -1.5708 + 0.9624i  -0.5236 + 0.0000i   0.5236 + 0.0000i   1.5708 - 0.9624i
```

Looking for matrix-functions, use `mfMul` for the (non commutative) multiplication. This routine is the equivalent of the `matmul` function in Fortran 90. On the other hand, the `dotproduct` Fortran 90 function doesn't need to exist in MUESLI: if you have two column vectors, use the transpose of the first vector multiplied by the second vector.

▶ Example #4-24:

```
  A = mfMagic(5)
  x = .t. [ 0, 0, 0, 1, 0 ] ! creation of a column vector
  call msDisplay(A,"A",x,"x")
  call msDisplay(mfMul(A,x),"mfMul(A,x)")! equiv. to the 'matmul' f90 intrinsic
```

▷ Output of ex. #4-24:

```
A =

   17    24     1     8    15
   23     5     7    14    16
    4     6    13    20    22
   10    12    19    21     3
   11    18    25     2     9

x =

    0
    0
    0
    1
    0

mfMul(A,x) =

    8
   14
   20
   21
    2
```

▶ Example #4-25:

```
  x = [ 1, 2, 3, 4, 5 ] ! creation of a two vectors
  y = [ 5, 4, 3, 2, 1 ]
  call msDisplay(x,"x",y,"y")
  ! equiv. to the 'dotproduct' f90 intrinsic
  call msDisplay(mfMul(x,.t.y),"dot product = mfMul(x,.t.y)")
```

▷ Output of ex. #4-25:

```
x =

    1     2     3     4     5


y =

    5     4     3     2     1


dot product = mfMul(x,.t.y) =

   35
```

### 4.2.4 Boolean `mfArrays`

Boolean **`mfArrays`** are automatically created when using comparison operators (`==`, `/=`, `>`, etc.). Syntactically, the same symbols than in Fortran 90 can be used, due to operator overloads. A similar situation arises with logical operators (`.not.`, `.and.`, `.or.`, etc.), and also for logical reduction functions

`All` and `Any`.

Arithmetic on boolean **mfArrays** is not permitted, but the matrix structure can be modified by the following routines: `mfReshape`, `mfRepMat`, `mfDiag` (only diagonal extraction from a matrix), `mfFlipLR`, `mfFlipUD`, `mfRot90` and the operators `.vc.`, `.hc.`, `.t.`.

▶ Example #4-26:

```
A = [ 1, 2, 3 ]
B = [ 3, 2, 1 ]
call msDisplay( A, "A", B, "B" )
call msDisplay( A==B, "A==B" ) ! boolean mfArray displayed
```

▷ Output of ex. #4-26:

```
A =

    1     2     3

B =

    3     2     1

A==B =

    F     T     F
```

▶ Example #4-27:

```
A = [ 1, 2, 3 ]
B = [ 3, 2, 1 ]
call msDisplay( A, "A", B, "B" )
call msDisplay( A>=B, "A>=B" ) ! boolean mfArray displayed
```

▷ Output of ex. #4-27:

```
A =

    1     2     3

B =

    3     2     1

A>=B =

    F     T     T
```

▶ Example #4-28:

```
A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay( A, "A" )
! by default, 'mfAll' works on columns
call msDisplay( mfAll(A>5.0d0), "mfAll(A>5.0d0)" )
call msDisplay( mfAll(A>5.0d0,1), "mfAll(A>5.0d0,1)" )
```

▷ Output of ex. #4-28:

```
A =

    1     4     7
    2     5     8
    3     6     9

mfAll(A>5.0d0) =

    F     F     T

mfAll(A>5.0d0,1) =

    F
    F
    F
```

To use boolean **mfArrays** in tests, the user should apply `All` or `Any` to them.

▶ Example #4-29:

```
A = reshape( [ (i, i = 1, 9) ], [ 3, 3 ] )
call msDisplay( A, "A" )
print *, "all(A>5.0d0) : ", all(A>5.0d0) ! logical result
if( any(A>=9.0d0) ) then
   print *, "at least one element of 'A' is greater or equal 9"
else
   print *, "no element of 'A' is greater or equal 9"
end if
```

▷ Output of ex. #4-29:

```
A =

    1     4     7
    2     5     8
    3     6     9

all(A>5.0d0) :  F
at least one element of 'A' is greater or equal 9
```

You can also assign classical Fortran logical arrays (rank 0, 1 or 2) to **mfArrays**.

▶ Example #4-30:

```
v = [ .true., .false., .true. ] ! creating a boolean vector
call msDisplay( v, "v" )
```

▷ Output of ex. #4-30:

```
v =

    T     F     T
```

Lastly, boolean **mfArrays** cannot have physical units (see section 4.3.2).

### 4.2.5 Doing Linear Algebra

There are a number of routines for doing linear algebra. Actually, they form an important part of the numerical part of MUESLI. Most of them are high-level routines which prepare data for a call to specific Lapack routines. So, solving a linear system of equations (`mfLDiv` or `mfRDiv`), factorizing a matrix in the product $LU$ (`msLU`), $QR$ (`msQR`) or getting its singular values (`msSVD`) can be realized via a single call, like in Matlab, with a very light and intuitive interface.

Only a subset of these routines is presented here; refer to the section "Matrix Functions" in the *MUESLI Reference Manual* for the complete list of available routines.

`mfLDiv` — the left division for matrices — is used to solve a linear system like $Ax = b$. It returns the vector solution $x = A^{-1}b$ without computing the inverse of the matrix $A$. Many internal tests are made to calling the most appropriate Lapack routine. The matrix $A$ doesn't need to have the same number of row and column: when facing with an overdetermined system, `mfLDiv` returns the least square solution; when facing with an undertermined system, `mfLDiv` returns one possible solution.

▶ Example #4-31:

```
n = 3
A = mfRand(n) + real(n)*mfEye(n)
b = .t. mf( [ (i, i=1,n) ] )
call msDisplay(A,"A",b,"b")
x = mfLDiv(A,b) ! solve the linear system : A.x = b    (x = A\b in Matlab)
call msDisplay(x,"x")
call msDisplay(mfNorm(mfMul(A,x)-b),"residue |A*x-b|")
```

▷ Output of ex. #4-31:

```
A =

   3.9839    0.6611    0.3045
   0.7000    3.8098    0.4843
   0.2753    0.9100    3.9854

b =

    1
    2
    3

x =

   0.1321
   0.4183
   0.6481

residue |A*x-b| =

    0
```

The previous routine always returns a solution, even if the matrix $A$ is singular. In the latter case, it firstly compute the pseudo-inverse of $A$ and then multiply it by $b$. Here is an example:

▶ Example #4-32:

```
n = 3 ! below, 'A' is a well-known example of singular matrix
A = mf( reshape( [ (i, i=1,n*n) ], [n,n] ) )
b = mfMul( A, mfOnes(n,1) ) ! taking 'b' in the image
call msDisplay(A,"A",b,"b")
x = mfLDiv(A,b) ! solution 'x' is not unique
call msDisplay(x,"x")
call msDisplay(mfNorm(mfMul(A,x)-b),"residue |A*x-b|")
```

▷ Output of ex. #4-32:

```
A =

    1     4     7
    2     5     8
    3     6     9

b =

    12
    15
    18

(MUESLI mfInv:) Warning: A is not inversible !
                Moore-Penrose pseudo-inverse will be returned

x =

   1.0000
   1.0000
   1.0000

residue |A*x-b| =

  8.1403E-15
```

In a similar way, the `mfRDiv` routine can be used to solve systems like $x\,A = b$ or $A'\,x = b'$; in the latter case, simply call the `mfRDiv` routine then transpose the returned vector, since it is equivalent to $x'\,A = b$.

The LU decomposition factorizes a matrix $A$ in two factors $L$ and $U$ such that $L\,U = A(p,:)$. Both factors are triangular matrices.

▶ Example #4-33:

```
A = mf( reshape( [ 1, 7, 1, 2, 8, 2, 3, 9, 4 ], [ 3, 3 ] ) )
call msDisplay(A,"A")
call msLU( mfOut(L,U,p), A ) ! output args are always enclosed in mfOut()
call msDisplay(L,"L",U,"U",p,"p")
call msDisplay( mfNorm(mfMul(L,U)-mfRowPerm(A,p)), "|L*U - A(p,:)|" )
```

▷ Output of ex. #4-33:

```
A =

    1     2     3
    7     8     9
    1     2     4

L =

   1.0000    0.0000    0.0000
   0.1429    1.0000    0.0000
   0.1429    1.0000    1.0000

U =

   7.0000    8.0000    9.0000
   0.0000    0.8571    1.7143
   0.0000    0.0000    1.0000

p =

    2
    1
    3

|L*U - A(p,:)| =

    0
```

The singular value decomposition of a matrix $A$ is obtained by calling the `msSvd` routine. This subroutine returns the three matrices $U, S, V$ such that $A = USV'$. Another function `mfSvd` only returns the singular values:

► Example #4-34:

```
A = .t. mf( reshape( [ (i,i=1,9) ], [ 3, 3 ] ) )
call msDisplay(A,"A")
! a singular matrix have at least one singular value equal to zero
call msDisplay( .t.mfDiag(mfSvd(A)), "S" ) ! displaying only the diagonal
! singular values are always stored in decreasing order
```

▷ Output of ex. #4-34:

```
A =

    1     2     3
    4     5     6
    7     8     9

S =

  16.8481    1.0684    0.0000
```

Other examples of linear algebra routines are `msChol`, `mfInv`, `mfNorm`, `mfCond` and `mfRank`.

### 4.2.6 Data Analysis with `mfArrays`

The following routines, whose names are evident to understand, are available for working with **`mfArrays`** : `mfMin`, `mfMax`, `mfSum`, `mfProd`, `mfSort`, `mfFind`, `mfMean`, `msMedian`, `mfVar`, `mfStd`, `mfRMS` and `msHist`. You can also apply the Fourier transform of your data in many variants: `mfFFT` (Fast Fourier Transform), `mfFourierCos` (Fourier-cosine), `mfFourierSin` (Fourier-sine), `mfFourierLeg` (Fourier-Legendre) and their inverse (resp. `mfInvFFT`, `mfInvFourierCos`, `mfInvFourierSin` and `mfInvFourierLeg`). Some of these routines have also a subroutine version, which allows generally a more sophisticated task. See the examples below.

▶ Example #4-35:

```
A = mfMagic(3)
call msDisplay(A,"A = mfMagic(3)")
call msDisplay(mfMin(A,1),"col min") ! returns a row vector
call msDisplay(mfMin(A,2),"row min") ! returns a column vector
! the scalar '4.0d0' is spread to the same shape than 'A'
call msDisplay(mfMin(A,4.0d0),"mfMin(A,4.0d0)")
```

▷ Output of ex. #4-35:

```
A = mfMagic(3) =

    8     1     6
    3     5     7
    4     9     2

col min =

    3     1     2

row min =

    1
    3
    2

mfMin(A,4.0d0) =

    4     1     4
    3     4     4
    4     4     2
```

▶ Example #4-36:

```
A = mfMagic(3)
call msDisplay(A,"A = mfMagic(3)")
! in a MUESLI subroutine call, 'mfOut' always encloses the output arguments
call msMax( mfOut(x,i), A, 1 ) ! works on columns
call msDisplay(x,"col max")
call msDisplay(i,"for index")
```

▷ Output of ex. #4-36:

```
A = mfMagic(3) =

    8     1     6
    3     5     7
    4     9     2

col max =

    8     9     7

for index =

    1     3     2
```

▶ Example #4-37:

```
  A = mfMagic(3)
  call msDisplay(A,"A = mfMagic(3)")
  call msDisplay(mfSum(A,1),"col sum")
  call msDisplay(mfSum(A,2),"row sum")
  call msDisplay(mfSum(mfDiag(A)),"diag. sum")
```

▷ Output of ex. #4-37:

```
A = mfMagic(3) =

    8     1     6
    3     5     7
    4     9     2

col sum =

   15    15    15

row sum =

   15
   15
   15

diag. sum =

   15
```

▶ Example #4-38:

```
  A = mfMagic(3)
  call msDisplay(A,"A = mfMagic(3)")
  call msDisplay(mfSort(A,1),"mfSort(A,1)") ! works on columns
```

▷ Output of ex. #4-38:

```
A = mfMagic(3) =

    8     1     6
    3     5     7
    4     9     2

mfSort(A,1) =

    3     1     2
    4     5     6
    8     9     7
```

▶ Example #4-39:

```
  A = mf( [ 2, 0, 0 ] ) .vc. &
      mf( [ 0, 0, 1 ] ) .vc. &
      mf( [ 0, 3, 0 ] )
  call msDisplay(A,"A")
  ! 'mfFind' returns indexes of non-zero elements
  long_ind = mfFind( A>1.0d0 ) ! works on a boolean mfArray
  call msDisplay(long_ind,"(long column indexes) mfFind( A>1.0d0 )")
```

▷ Output of ex. #4-39:

```
A =

    2     0     0
    0     0     1
    0     3     0


(long column indexes) mfFind( A>1.0d0 ) =

    1     6
```

You can also use the `msFind` routine to directly get the $(i, j)$ indexes instead of long column indexes.

MUESLI also provides some polynomial routines essentially to make interpolation. Firstly, cubic spline interpolation is available, via `mfSpline` and `mfPPVal`. This is a 1-D interpolation.

▶ Example #4-40:

```
  x = [ 0.0d0, 0.5d0, 1.0d0 ]
  y = [ 0.0d0, 1.0d0, 0.0d0 ]
  call msDisplay(x,"x",y,"y")
  pp = mfSpline( x, y )
  ! 'pp' actually contains the second derivatives of 'y' with respect to 'x'
  call msDisplay( pp, "pp = mfSpline( x, y )" )
```

▷ Output of ex. #4-40:

```
x =

   0.0000    0.5000    1.0000

y =

    0     1     0

pp = mfSpline( x, y ) =

    0   -12     0
```

▶ Example #4-41:

```
  xx = mfLinspace( -0.25d0, 1.25d0, 7 )
  call msDisplay(xx, "xx")
  ! 'x', 'y' and 'pp' comes from the previous example
  yy = mfPPVal( x, y, pp, xx )
  call msDisplay( yy, "yy = mfPPVal( x, y, pp, xx )" )
```

▷ Output of ex. #4-41:

```
xx =

  -0.2500    0.0000    0.2500    0.5000    0.7500    1.0000    1.2500

yy = mfPPVal( x, y, pp, xx ) =

  -0.6875    0.0000    0.6875    1.0000    0.6875    0.0000   -0.6875
```

Secondly, there are also 2-D interpolation routines, either for a regular grid (`mfInterp2`) or for an irregular grid (`mfGridData`).

For `mfInterp2`, the interpolation method is bilinear with respect to the two coordinates $(x,y)$. For example, coordinates and data are stored in rank-2 `mfArrays` `x`, `y`, `z`. `x` and `y` can be obtained from the `mfMeshGrid` routine. The output may be either a single point, either a vector of points, either a matrix of points according the shape of the `mfArrays` `xi` and `yi`.

▶ Example #4-42:

```
  a = mfLinspace( 0.0d0, 1.0d0, 4 ); b = mfLinspace( 0.0d0, 1.0d0, 3 )
  call msMeshgrid( mfOut(x,y), a, b )
  z = x
  xi = mfColon( -0.2d0, 0.2d0, 1.2001d0 ); yi = xi ! vector inputs
  call msDisplay( x, "x", y, "y", xi, "xi", yi, "yi" )
  zi = mfInterp2( x, y, z, xi, yi ) ! output is a vector
  print *, "NaN are obtained for 'out-of-range' input points."
  call msDisplay( zi, "interpolated data at points on a line" )
```

▷ Output of ex. #4-42:

```
x =

   0.0000    0.3333    0.6667    1.0000
   0.0000    0.3333    0.6667    1.0000
   0.0000    0.3333    0.6667    1.0000

y =

   0.0000    0.0000    0.0000    0.0000
   0.5000    0.5000    0.5000    0.5000
   1.0000    1.0000    1.0000    1.0000

xi =

  -0.2000    0.0000    0.2000    0.4000    0.6000    0.8000    1.0000    1.2000

yi =

  -0.2000    0.0000    0.2000    0.4000    0.6000    0.8000    1.0000    1.2000

NaN are obtained for 'out-of-range' input points.

interpolated data at points on a line =

     NaN    0.0000    0.2000    0.4000    0.6000    0.8000    1.0000       NaN
```

mfGridData considers three vectors of type **mfArray**, which contain respectively the coordinates of irregular points and the attached values. It interpolates with a linear scheme.

▶ Example #4-43:

```
x = [ -0.15d0,  1.12d0, 1.10d0, -0.12d0, 0.5d0 ]
y = [ -0.11d0, -0.13d0, 1.14d0,  1.13d0, 0.5d0 ]
z = x
call msDisplay( .t.((x .vc. y) .vc. z), "nodes and values" )
xi = 0.1234d0 ! scalar input
yi = 0.9876d0
call msDisplay( xi, "xi", yi, "yi" )
zi = mfGridData( x, y, z, xi, yi ) ! output is a scalar
call msDisplay( zi, "interpolated data at a point" )
```

▷ Output of ex. #4-43:

```
nodes and values =

  -0.1500   -0.1100   -0.1500
   1.1200   -0.1300    1.1200
   1.1000    1.1400    1.1000
  -0.1200    1.1300   -0.1200
   0.5000    0.5000    0.5000

xi =

   0.1234

yi =

   0.9876

interpolated data at a point =

   0.1234
```

Lastly, fitting (both for polynomial or any user-supplied function) may be easily done by using the routines `msPolyFit` and `msFunFit`.

Other routines exist in order to build a Delaunay triangulation and to search for nodes or triangles inside this triangulation. We plan to make a routine for building a plane triangulation inside specified boundaries.

### 4.2.7 Other numerical tasks

With MUESLI, you can easily perform zero finding of functions (`mfFZero` for the single-variable case; `mfFSolve` for the multiple-variable case), nonlinear least-square minimization (`mfLsqNonLin`), integration of data (*i. e.* function integration or quadrature schemes) and integration of systems of ODEs and DAEs (*resp.* Ordinary Differential Equations and Differential Algebraic Equations).

The `mfFZero` routine searches by bissection the zero of a function given by the user:

▶ Example #4-44:

```
! definition of a user-function (actually in an external module)
function fun_sqr_minus_onehalf( x ) result( res )
   real(kind=MF_DOUBLE), intent(in) :: x
   real(kind=MF_DOUBLE) :: res
   res = x**2 - 0.5d0
end function fun_sqr_minus_onehalf
!-----------
res = mfFZero( fun_sqr_minus_onehalf, x0=mf([ 0, 1 ]) )
call msDisplay( res, "numerical result" )
```

▷ Output of ex. #4-44:

```
numerical result =

   0.7071
```

The mfQuad routine computes the numerical integration on a function, which must be supplied by the user. Let's compute, for example, the quantity:

$$\int_0^1 \sqrt{x}\, dx$$

▶ Example #4-45:

```
! definition of a user-function (actually in an external module)
function fun_sqrt( x ) result( res )
   real(kind=MF_DOUBLE), intent(in) :: x
   real(kind=MF_DOUBLE) :: res
   res = sqrt( x )
end function
!-----------
res = mfQuad( fun_sqrt, 0.0d0, 1.0d0 ) ! integration of 'sqrt(x)' over [0,1]
call msDisplay( res, "numerical result" )
```

▷ Output of ex. #4-45:

```
numerical result =

   0.6667
```

Therefore

$$\int_0^1 \sqrt{x}\, dx \approx 0.6667$$

On the other hand, msQuad allows the user to get more information returned by the integrator. Beside, MUESLI offers two other basic integrators: mfTrapz and mfSimpson.

The mfLsqNonLin routine find the parameters which minimize a function. Let's consider the following module:

▶ Example #4-46:

```
module optim_funs
   use fml
   implicit none
contains
   subroutine exp_min( m, n, p, fvec, flag )
      integer,               intent(in) :: m, n
      real(kind=MF_DOUBLE), intent(in) :: p(n)
      real(kind=MF_DOUBLE)             :: fvec(m)
      integer                          :: flag
      !---
      integer :: i
      if( p(1) < 0.0d0 ) then
         flag = +1 ! out-of-range condition to avoid an invalid result in sqrt
         return
      end if
      do i = 1, m
         fvec(i) = 2.0d0*i + i*(1.0d0-p(1)) + sqrt(i*p(1)) + exp(-i*p(2))
      end do
   end subroutine exp_min
end module optim_funs
```

The previous module defines the function to be minimized, `exp_min`, with respect to two parameters `p(1)` and `p(2)`. Note the presence of the `sqrt` function which is not defined for a negative parameter p(1); this is why we added a returned flag equal to +1 when `p(1)` is found to be out-of-range (in other words, a constraint on `p(1)` is included in `fcn`). The call to the `mfLsqNonLin` routine could be written as simply as:

▶ Example #4-47:

```
m = 10
n = 2
z = mfLsqNonLin( m, exp_min, p, n )
call msDisplay( z, "solution" )
```

▷ Output of ex. #4-47:

```
(MUESLI mfLsqNonLin:) Warning: initial guess 'p0' is empty: trying to use
                      a zero vector!
(MUESLI mfLsqNonLin:) ERROR: [minpack]/lmdif1 routine failed.
                      for your information: info = -16
```

However, as seen above, the previous call will returned an error ($-16$, which means that a too large number of reducing the step bound has been used in the algorithm, according the table of errors of `msLsqNonLin` in the *Muesli Reference Manual*. The next call will lead to a solution, because, the initial guess is not too close the critical limit of definition of the `sqrt` function.

▶ Example #4-48:

```
m = 10
n = 2
p = [ 0.5d0, -0.5d0 ] ! initial guess
z = mfLsqNonLin( m, exp_min, p, n )
call msDisplay( z, "solution" )
```

▷ Output of ex. #4-48:

```
solution =

   4.6201   -0.2304
```

The `mfOdeSolve` routine solves an ODE system described by a user subroutine. A second order differential equation has been chosen for this purpose:

$$\frac{\partial^2 y(x)}{\partial x^2} = -y(x), \quad x > 0\,; \qquad y(0) = 0, \quad y'(0) = 1$$

which must be decomposed into the following system of two first order differential equations:

$$\begin{aligned} y_1'(x) &= y_2(x) \\ y_2'(x) &= -y_1(x) \end{aligned}$$

Then, this system is in turn described in the `deriv` following subroutine.

▶ Example #4-49:

```
! definition of a user-subroutine (actually in an external module)
subroutine deriv( t, y, yprime )
   real(kind=MF_DOUBLE), intent(in) :: t, y(*)
   real(kind=MF_DOUBLE)             :: yprime(*)
   yprime(1) =  y(2)
   yprime(2) = -y(1)
end subroutine
!-----------------------------
! type(mfArray) :: t_0, t_end
print *, "integration of {y'' = - y} over [0,pi]:"
print *, "  [ solution is: y(t) = sin(x) ]"
t_0 = 0.0d0
t_end = MF_PI
t_span = .t. mfLinSpace( t_0, t_end, 10 ) ! 10 values required for output
y = [ sin(t_0), cos(t_0) ]
y_end = mfOdeSolve( deriv, t_span, y )
call msDisplay( y_end, "numerical result" ) ! Runge-Kutta Fehlberg
```

▷ Output of ex. #4-49:

```
integration of {y'' = - y} over [0,pi]:
  [ solution is: y(t) = sin(x) ]

numerical result =

   0.0000    1.0000
   0.3420    0.9397
   0.6428    0.7660
   0.8660    0.5000
   0.9848    0.1737
   0.9848   -0.1736
   0.8660   -0.5000
   0.6428   -0.7661
   0.3420   -0.9397
   0.0000   -1.0000
```

By default, The *Runge-Kutta Fehlberg* method is used. It is suitable for most of ODE systems, except for stiff systems. Other available methods are *Adams-Bashforth-Moulton* (non-stiff systems) and *Backward Differentiation Formula* (stiff systems). Again, `msOdeSolve` allows the user to get more information returned by the ODE solver.

For implicit ODE systems (or DAE systems), you can use the `mfDaeSolve` routine, which has an interface similar to `mfOdeSolve`. But be aware that the current implementation can deal with DAE systems of differential index 1 only (this can be changed in the future...).

### 4.2.8 Saving/reading `mfArrays` to/from disk

Saving data in a file is important for real applications. MUESLI provides several ways to save your `mfArrays`.

The most simple way is to save one `mfArray` in an ASCII file: `msSaveAscii` copies only the real part, and formats it under an array form. It is somewhat slow and not very efficient, because ASCII data leads to large file size. However, ASCII files can be read by any application on any platform, therefore it is a great advantage. The companion routine `mfLoadAscii`, reads an ASCII file (usually organized like a matrix but it is not strictly required), and stores it in an `mfArray`.

If your are looking for smaller file size, another way is to write data in a binary format. `msSave` copies an `mfArray` in a binary file using a special format (specific to the MUESLI library), because it stores also other information about the type of data, its structure and some matrix properties. As a consequence, the created file may be read only by the `msLoad` routine. The proposed extension for this kind of file is `.mbf` (MUESLI Binary File), but it is not strictly required because the binary file contains a special tag to be automatically recognized. Lastly, `msSave` can even write directly in a gzipped binary file: just provide a name which end with `.mbf.gz`!

MUESLI provides also two routines, `msSaveHDF5` and `mfLoadHDF5`, to write and read `mfArrays` in HDF5 files. Many `mfArrays` can be stored in a single HDF5 file but currently in a flat way, not yet in a hierarchical way as possible in the HDF5 format.

See also in section 4.2.9 about sparse matrices to know how to save them in files, because MUESLI proposes other specific routines.

### 4.2.9 Sparse `mfArrays`

Sparse structure for arrays are used when the number of non-zero elements is small in comparison with the total number of elements. This technique allows to reduce both memory space and floating-point operations. MUESLI uses internally the CSC format (Compact Sparse Columns), and all routines involving sparse `mfArrays` are of course adapted to this special storage; on the other hand, MUESLI is able to read (from files) and import (from ordinary f90 arrays) other sparse format, such as COO (Coordinates) and CSR (Compact Sparse Row).

Sparse `mfArrays` doesn't arise automatically: the programmer has to create initially sparse arrays or has to convert explicitly dense `mfArrays` to sparse ones. Beside the two basic routines: `mfSpAlloc` (which create a new sparse `mfArray`) and `mfSparse` (which convert a dense `mfArray`), other specific routines are useful: `mfSpEye` (creates a sparse identity matrix) and `mfSpDiags` (create a sparse diagonal matrix from a vector).

▶ Example #4-50:

```
A = mfSpEye( 5 )
call msDisplay(A, "mfSpEye( 5 )")
```

▷ Output of ex. #4-50:

```
mfSpEye( 5 ) =

    real sparse matrix of size : 5, 5
    nz : 5, nzmax : 5
    (1,1)    1.0000E+00
    (2,2)    1.0000E+00
    (3,3)    1.0000E+00
    (4,4)    1.0000E+00
    (5,5)    1.0000E+00
```

To build a new sparse `mfArray`, you can also aggregate previously existing `mfArrays` using the concatenation operators `.vc.` and `.hc.` or, better, the routine `msHorizConcat`, as in the following example:

▶ Example #4-51:

```
! building a big (nrow,ncol) sparse mfArray from its columns
A = mfSpAlloc( nrow, 0 ) ! A is sparse but has zero columns
do i = 1, ncol
! filling the dense column vector vec_col ...
   call msHorizConcat( A, vec_col ) ! 'in place' operation
end do
```

If you don't have easily access to columns of the sparse matrix you want to build but only the rows, you can perform the same thing but in a transpose mode (because `msVertConcat` doesn't exist):

▶ Example #4-52:

```
! building a big (nrow,ncol) sparse mfArray from its rows
A = mfSpAlloc( ncol, 0 ) ! A is sparse but has zero columns
do i = 1, nrow
! filling the dense row vector vec_row ...
   call msHorizConcat( A, vec_row ) ! 'in place' operation
end do
! lastly, transpose the mfArray
A = .t. A
```

When working with sparse matrices, the routine `msDisplay` prints the actual size of the array, the number of non-zero values and the maximum number of elements. This last value can be modified (up or down) via the routine `msSpReAlloc`. You can also pack your sparse matrix to its minimum size via the same routine, by using the `"minimal"` argument (see the *MUESLI Reference Manual* document).

▶ Example #4-53:

```
v = .t. [ 1.0d0, 2.0d0, 3.0d0 ]
call msDisplay(v,"v")
B = mfSpDiags( 3, 5, v, 0 ) ! last arg. : zero-diagonal is the main diagonal
call msDisplay(B,"B = mfSpDiags( 3, 5, v, 0 )")
```

▷ Output of ex. #4-53:

```
v =

    1
    2
    3

B = mfSpDiags( 3, 5, v, 0 ) =

    real sparse matrix of size : 3, 5
    nz : 3, nzmax : 3
    (1,1)    1.0000E+00
    (2,2)    2.0000E+00
    (3,3)    3.0000E+00
```

▶ Example #4-54:

```
A = mfSpAlloc(10,10,nzmax=30) ! used to reserve storage place in advance
print *, "A = mfSpAlloc(10,10,nzmax=30)"
call msDisplay(mfShape(A), "mfShape(A)")
call msDisplay(mfSize(A), "mfSize(A)")
call msDisplay(mfNzmax(A), "mfNzmax(A)")
call msDisplay(mfNnz(A), "mfNnz(A)")
```

▷ Output of ex. #4-54:

```
A = mfSpAlloc(10,10,nzmax=30)

mfShape(A) =

   10    10

mfSize(A) =

  100

mfNzmax(A) =

   30

mfNnz(A) =

    0
```

▶ Example #4-55:

```
A = mfEye(5,5)
call msSet( -1.0d0, A, 1, 5 )
call msSet( -1.0d0, A, 5, 1 )
! A has only 7 non zero element over 25
B = mfSparse(A)
call msDisplay(B, "B = mfSparse(A)")
```

▷ Output of ex. #4-55:

```
B = mfSparse(A) =

    real sparse matrix of size : 5, 5
    nz : 7, nzmax : 7
    (1,1)    1.0000E+00
    (5,1)   -1.0000E+00
    (2,2)    1.0000E+00
    (3,3)    1.0000E+00
    (4,4)    1.0000E+00
    (1,5)   -1.0000E+00
    (5,5)    1.0000E+00
```

MUESLI also provides exchange routine: `mfSpImport` and `msSpExport` which support many format (COO, CSC and CSR).

▶ Example #4-56:

```
! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(3), jc(3), val(3) )
ir(:) = [ 2, 3, 4 ]
jc(:) = [ 2, 4, 3 ]
val(:) = [ 1.0d0, 2.0d0, 3.0d0 ]
A = mfSpImport(ir,jc,val) ! default: COO format is used
call msDisplay(A, "A = mfSpImport(ir,jc,val)")
```

▷ Output of ex. #4-56:

```
A = mfSpImport(ir,jc,val) =

   real sparse matrix of size : 4, 4
   nz : 3, nzmax : 3
   (2,2)    1.0000E+00
   (4,3)    3.0000E+00
   (3,4)    2.0000E+00
```

▶ Example #4-57:

```
! integer, allocatable :: ptr(:), j(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

A = mf( [ 0, 3, 4, 0, 0 ] ) .vc.                                    &
    mf( [ 1, 0, 0, 6, 0 ] ) .vc.                                    &
    mf( [ 0, 0, 0, 0, 7 ] ) .vc.                                    &
    mf( [ 2, 0, 5, 0, 0 ] )

call msDisplay(A,"A")
A = mfSparse(A)

nrow = mfGet( mfShape(A), 1 )
nnz = mfNnz(A)
allocate( ptr(nrow+1), j(nnz), val(nnz) )
call msSpExport( A, ptr, j, val, format="CSR" )
print *, "exporting 'A' to CSR format:"
print *, "  ptr = ", ptr
print *, "  j   = ", j
print *, "  val = ", val
```

▷ Output of ex. #4-57:

```
A =

    0     3     4     0     0
    1     0     0     6     0
    0     0     0     0     7
    2     0     5     0     0

exporting 'A' to CSR format:
  ptr =  1 3 5 6 8
  j   =  2 3 1 4 5 1 3
  val =  3. 4. 1. 6. 7. 2. 5.
```

To modify an element of a sparse `mfArray` you can use, as for a dense `mfArray`, the routine `msSet`. However, it must be emphasized that modifying a single value (especially when the entry doesn't exist) is very inefficient; the following strategy should then be used instead: get the corresponding column, change the value (or better, update all values which must be changed) and put back the modified column in the `mfArray x`. Lastly, modifying a whole column is more efficient than modifying a whole row.

Additionally, `mfFull` is used to convert from sparse to dense.

Nearly all arithmetic operations and most of routines can be applied transparently to sparse `mfArray`s. In the contrary, you will be warned by a specific message. For example, solving a sparse linear system of equation is done by the usual way (as already seen in section 4.2.5):

▶ Example #4-58:

```
! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(12), jc(12), val(12) )
ir(:) = [ 1, 2, 1, 3, 5, 2, 3, 4, 5, 3, 2, 5 ]
jc(:) = [ 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 5, 5 ]
val(:) = [ 2.0d0, 3.0d0, 3.0d0, -1.0d0, 4.0d0, 4.0d0, -3.0d0,        &
           1.0d0, 2.0d0, 2.0d0,  6.0d0, 1.0d0 ]

A = mfSpImport(ir,jc,val)
call msDisplay(mfFull(A), "Sparse matrix A")

b = mfMul( A, .t. mf( [ 1, 2, 3, 4, 5 ] ) )
call msDisplay(b, "b [RHS]")

x = mfLDiv( A, b ) ! for sparse systems, UMFPack library is used
call msDisplay(x, "x [sol.]")
```

▷ Output of ex. #4-58:

```
Sparse matrix A =

    2     3     0     0     0
    3     0     4     0     6
    0    -1    -3     2     0
    0     0     1     0     0
    0     4     2     0     1

b [RHS] =

    8
   45
   -3
    3
   19

x [sol.] =

   1.0000
   2.0000
   3.0000
   4.0000
   5.0000
```

Generally operations between mixed storage structures are not allowed (you must use the same storage structure), except for the matrix-vector product, which can be sparse/dense, and few others.

If you have any doubt about the sparsity of an `mfArray`, use the inquiry logical functions `mfIsSparse` and `mfIsDense`.

▶ Example #4-59:

```
A = mfSpEye(1)
print *, "mfIsSparse(mfSpEye(1)) = ", mfIsSparse(A)
```

▷ Output of ex. #4-59:

```
mfIsSparse(mfSpEye(1)) =  T
```

Sparse matrices have generally a big number of rows and columns. So, when computing eigenvalues or singular values, we like to find not all values, but only some of them. This can be done with `mfEigs` and `mfSVDS` (if you obtain some convergence problems, you should use the '`ncv=`' option):

▶ Example #4-60:

```
! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(9), jc(9), val(9) )
ir(:) = [ 1, 1, 2, 2, 3, 4, 4, 5, 5 ]
jc(:) = [ 1, 5, 2, 4, 3, 2, 4, 1, 5 ]
val(:) = [ 2.0d0, 1.0d0, 1.0d0, -3.0d0, 1.0d0, -3.0d0, 1.0d0, 1.0d0, 2.0d0 ]

A = mfSpImport(ir,jc,val)
call msDisplay(mfFull(A), "Sparse matrix A")

d = mfEigs( A, 3 ) ! ARPACK is used (which ? Largest Magnitude by default)
call msDisplay(d, "Largest eigenvalues from mfEigs()")
```

▷ Output of ex. #4-60:

```
Sparse matrix A =

    2     0     0     0     1
    0     1     0    -3     0
    0     0     1     0     0
    0    -3     0     1     0
    1     0     0     0     2

Largest eigenvalues from mfEigs() =

   4.0000
   3.0000
  -2.0000
```

▶ Example #4-61:

```
! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(9), jc(9), val(9) )
ir(:) = [ 1, 1, 2, 2, 3, 4, 4, 5, 5 ]
jc(:) = [ 1, 5, 2, 4, 3, 2, 4, 1, 5 ]
val(:) = [ 2.0d0, -1.0d0, 1.0d0, -3.0d0, 1.0d0, 3.0d0, 1.0d0, 1.0d0, 2.0d0 ]

A = mfSpImport(ir,jc,val)
call msDisplay(mfFull(A), "Sparse matrix A")

d = mfSVDS( A, 3, which="SM" ) ! which ? Smallest Magnitude
call msDisplay(d, "Smallest singular values from mfSVDS()")
```

▷ Output of ex. #4-61:

```
Sparse matrix A =

    2     0     0     0    -1
    0     1     0    -3     0
    0     0     1     0     0
    0     3     0     1     0
    1     0     0     0     2

Smallest singular values from mfSVDS() =

   1.0000
   2.2361
   2.2361
```

To perform IO operations, both `msSave` and `mfLoad` (described in section 4.2.8) support sparse `mfArrays`. Moreover, for ASCII format, there are `msSaveSparse` and `mfLoadSparse`, using different ASCII formatting:

- `"CSC"` for "Compact Sparse Column" (default)
- `"CSR"` for "Compact Sparse Row"
- `"HBO"` for "Harwell-Boeing"
- `"MTX"` for "Matrix Market"

The first two formats "CSC" or "CSR", correspond to a write of a first line of 5 objects, then the write of the three vectors: pointer to indexes, indexes and values. See the following example:

▶ Example #4-62:

```
! integer, allocatable :: ir(:), jc(:)
! real(kind=MF_DOUBLE), allocatable :: val(:)

allocate( ir(9), jc(9), val(9) )
ir(:) = [ 5, 1, 1, 2, 2, 3, 4, 4, 5 ]
jc(:) = [ 1, 1, 5, 2, 4, 3, 2, 4, 5 ]
val(:) = [ 8.0d0, 1.0d0, 2.0d0, 3.0d0, 4.0d0, 5.0d0, 6.0d0, 7.0d0, 9.0d0 ]

A = mfSpImport(ir,jc,val)
call msDisplay(mfFull(A),"Sparse Matrix A")
call msSaveSparse( "A.csc", A, format="CSC" )
```

▷ Output of ex. #4-62:

```
Sparse Matrix A =

    1     0     0     0     2
    0     3     0     4     0
    0     0     5     0     0
    0     6     0     7     0
    8     0     0     0     9
```

Then, the file "A.csc" contains the following data:

```
5 5 -1 csc real
  1  3  5  6  8 10
 5 1 2 4 3 2 4 1 5
  8.000000000000000E+00  1.000000000000000E+00  3.000000000000000E+00
  6.000000000000000E+00  5.000000000000000E+00  4.000000000000000E+00
  7.000000000000000E+00  2.000000000000000E+00  9.000000000000000E+00
```

The first line of the previous file contains the shape of the sparse matrix $(5, 5)$, followed by the "row sorted" property (here $-1$, because this property is unknown), then a keyword describing the compact format used ("csc" or "csr") and last the type of the data ("real" or "complex").

Matrix Market format ("MTX") uses the coordinates format ("COO").

Harwell-Boeing format ("HBO") is very similar to "CSC" or "CSR" format, except that the header is different and it is more compact because there is no blanks between indexes and numbers; as a consequence, this format is not always suitable for a human read.

The internal sparse format used (CSC format) doesn't need the elements to be row sorted in each column. However, some sparse algorithms are more efficient when columns of sparse matrices are row sorted. By default, MUESLI automatically sorts the rows when needed; use `msSetAutoRowSorted` to change this default behavior. See also `mfIsRowSorted` and `msRowSort` in the *MUESLI Reference Manual*.

## 4.3 Advanced topics

### 4.3.1 Matrix Properties

We have seen in section 4.2.5 that some high-level linear algebra routines check for some properties of an `mfArray` before applying the specific Lapack routine. These properties are the matrix symmetry and its definite positiveness (which are theoretically not dependent). Sometimes, the user may find useful to get these properties, which are stored in the internal structure of the `mfArray`. To do that, two inquiry functions are available: `mfIsSymm` and `mfIsPosDef`.

▶ Example #4-63:

```
A = mfHilb(5) ! Hilbert matrices are always symmetric
print *, "Is A symmetric ? ", mfIsSymm( A )
B = mfMagic(5) ! Magic squares are never symmetric
print *, "Is B symmetric ? ", mfIsSymm( B )
```

▷ Output of ex. #4-63:

```
Is A symmetric ?  T
Is B symmetric ?  F
```

▶ Example #4-64:

```
A = mfHilb(5) ! Hilbert matrices are always positive definite
print *, "Is A positive definite ? ", mfIsPosDef( A )
B = mfOnes(5)
print *, "Is B positive definite ? ", mfIsPosDef( B )
```

▷ Output of ex. #4-64:

```
Is A positive definite ?  T
Is B positive definite ?  F
```

Sparse `mfArrays` have an additional property (see section 4.2.9).

### 4.3.2 Physical Units in `mfArrays`

When doing some computations, a physicist manipulates only numerical quantities and looses the physical units. Most of time, he has prepared his equations before and knows exactly what he has coded. Unfortunately, if his equations are not physically consistent, this will cannot be detected at run-time and the results will be wrong.

MUESLI lets the user to assign each `mfArray` (not of type boolean) a physical unit. These physical units are composed together during run-time execution and any physical inconsistency will be detected and signaled as soon as it arises.

By default, MUESLI doesn't take physical units into consideration. The user must call the `msUsePhysUnits` routine to make them active.

The `msDisplay` routine takes care of the physical unit, if present, and prints inside brackets the decomposition of this unit into the seven following fundamental units:

- `M` (mass)
- `L` (length)
- `T` (time)
- `theta` (temperature)
- `I` (electric current intensity)
- `mol` (mole)
- `l` (luminous intensity)

▶ Example #4-65:

```
call msUsePhysUnits( "on" ) ! default is "off"
x = 2.0d0
call msSetPhysDim( x, Mass=1.0d0/2, Length=-7.0d0/2, Time=1.0d0/3 )
call msDisplay( x, "x" )
```

▷ Output of ex. #4-65:

```
x =
[ M^(1/2) L^(-7/2) T^(1/3) ]

    2
```

Predefined common units are useful to set physical unit in an easier way:

▶ Example #4-66:

```
! type(mfUnit) :: u_Ly
call msUsePhysUnits( "on" )
y = 0.25d0*u_kg ! 'u_kg' is a predefined unit
call msDisplay( y, "y" )

u_Ly = u_calorie / u_cm**2 / u_min ! new user-defined unit
call msSetPhysUnitAbbrev( u_Ly, "Langley" )

x = 1.95d0 * u_Ly
call msDisplay( x, "x", unit=u_Ly )
call msDisplay( x, "", unit=SI_unit )
```

▷ Output of ex. #4-66:

```
y =
[ M ]

   0.2500

x =
[ M T^-3 ] (Langley)

   1.9500

 =
[ M T^-3 ] (S.I.)

  1.3585E+03
```

As seen in the previous example, you can combine predefined units in order to build new ones. You can even attach the abbrevation of your choice to the new user-defined `mfUnit`. When combining `mfUnit`s, the following operators can be used: `*`, `/` and `**` (integer or real exponent). When using the exponentiation operator (`**`) with a real exponent, be aware that the exponent will be converted to a 2-bytes based rational number: this means that `0.33333d0` will be replaced by 1/3 but that `0.3333d0` will be replaced by 3333/10000; in this later example, the best practice is to employ `(1.0d0/3.0d0)` as exponent.

Hereafter is the list of all predefined variables (units, multipliers, constants and conversion factors):

– predefined units begin with the `u_` prefix:

`u_kg` = `u_kilogram` : kilogram
`u_m` = `u_meter` : meter
`u_s` = `u_second` : second
`u_K` = `u_Kelvin` : Kelvin
`u_A` = `u_Ampere` : Ampere
`u_mol` = `u_mole` : mole
`u_cd` = `u_candela` : candela

`u_mg` : milligram
`u_cg` : centigram
`u_g` : gram
`u_angstrom` : angström
`u_micron` : micrometer
`u_mm` : millimeter
`u_cm` : centimeter
`u_km` : kilometer

`u_astronomical_unit` : astronomical unit
`u_min` = `u_minute` : minute ($\frac{1}{60}$ hour)
`u_hr` = `u_hour` : hour
`u_day` : day
`u_liter` : liter
`u_hertz` : Hertz
`u_Newton` : Newton
`u_Pascal` : Pascal
`u_millimeter_Hg` : millimeter Hg
`u_bar` : bar
`u_atm` : atmosphere
`u_psi` : psi (pound by square inch)
`u_Joule` : Joule
`u_calorie` : calorie
`u_BTU` : BTU (British thermal unit)
`u_Watt` : Watt
`u_Poiseuille` : Poiseuille
`u_Coulomb` : Coulomb
`u_Volt` : Volt
`u_Farad` : Farad
`u_Ohm` : Ohm
`u_Weber` : Weber
`u_Tesla` : Tesla
`u_Henry` : Henry

– multipliers begin with the `m_` prefix:

`m_yotta` = $10^{24}$
`m_zetta` = $10^{21}$
`m_exa` = $10^{18}$
`m_peta` = $10^{15}$
`m_tera` = $10^{12}$
`m_giga` = $10^{9}$
`m_mega` = $10^{6}$
`m_kilo` = $10^{3}$
`m_hecto` = $10^{2}$
`m_deca` = $10^{1}$
`m_deci` = $10^{-1}$
`m_centi` = $10^{-2}$
`m_milli` = $10^{-3}$
`m_micro` = $10^{-6}$
`m_nano` = $10^{-9}$
`m_pico` = $10^{-12}$
`m_femto` = $10^{-15}$
`m_atto` = $10^{-18}$
`m_zepto` = $10^{-21}$
`m_yocto` = $10^{-24}$

– physical constants begin with the `c_` prefix:

`c_speed_of_light` : speed of light $[LT^{-1}]$
`c_Planck` : Planck constant $[ML^2T^{-1}]$
`c_Avogadro` : Avogadro number $[mol^{-1}]$
`c_universal_gas` : universal gas constant $[ML^2T^{-2}\Theta^{-1}mol^{-1}]$
`c_Boltzmann` : Boltzmann constant $[ML^2T^{-2}\Theta^{-1}]$
`c_electron_charge` : electron charge $[TI]$
`c_gravity` : gravity constant $[M^{-1}L^3T^{-2}]$
`c_gravity_accel` : gravity acceleration $[LT^{-2}]$

– conversion factors begin with the `f_` prefix:

`f_kelvin_conversion` : temperature of triple point of water [Θ]

When printing a physical quantity, you may change its displayed unit. Note that to avoid confusion, the name of the required unit is added in parenthesis. If you don't known the S.I. name of a quantity, you can let MUESLI try to find it, by using '`unit=SI_unit`' as additional argument to `msDisplay`

▶ Example #4-67:

```
L = u_m ! length
call msDisplay( L, "length", unit=u_cm ) ! should be 100 centimeters
F = u_Newton ! force
call msDisplay( F, "force", unit=SI_unit )
A = L*L ! area
call msDisplay( A, "area", unit=SI_unit )
P = F/A ! pressure
call msDisplay( P, "pressure", unit=SI_unit )
```

▷ Output of ex. #4-67:

```
length =
[ L ] (cm)

  100

force =
[ M L T^-2 ] (N)

    1

area =
[ L^2 ] (S.I.)

    1

pressure =
[ M L^-1 T^-2 ] (Pa)

    1
```

Finally, you can check whether a quantity is dimensionless, or whether two quantities share the same physical dimensions:

▶ Example #4-68:

```
x = 2.0d0 * u_m
y = 3.5d0 * u_cm
print *, "Do 'x' and 'y' share the same phys. dim. ? ", mfHaveSamePhysDim(z)
z = x/y
print *, "Is 'z' dimensionless ? ", mfHasNoPhysDim(z)
```

▷ Output of ex. #4-68:

```
Do 'x' and 'y' share the same phys. dim. ? T
Is 'z' dimensionless ? T
```

### 4.3.3 Performance issues

#### Avoiding copy of a temporary array

When you use, for example, the following form of assignment:

```
A = mfRand(N,N)
```

MUESLI first creates a temporary array, in which it stores the random values and then, copy this array into `A`. You can be warned of this copy by setting the message level to an appropriate level, *i. e.* inserting the following in your source code:

```
call msSetMsgLevel(3)
```

In such a case, you will see as output:

```
(MUESLI assignment(=):) info: you are using the simple form of
                        assignment 'a=b', which copy the RHS tempo values.
                        You could use the other form : 'call msAssign(a,b)',
                        which only points to data, and therefore is much more
                        efficient !
```

(you cannot simply locate the concerned statement, because the routine `msSetTrbLevel` doesn't apply to the *info* level). To improve performance, you may use the subroutine form of the assignment:

```
call msAssign( A, mfRand(N,N) )
```

which avoid such a copy. More explanation can be found in the *MUESLI inside* document.

#### Counting Floating-Point Operations

Counting Floating-Point Operations (or, in short, flops) is possible in MUESLI only via hardware counters. Therefore, you must have some software installed to access these counters. On linux, at least for the 2.4 and 2.6 series, the kernel must be patched with 'perfctr' and the PAPI library must be installed.

If your system contains the PAPI library, don't forget to tell to MUESLI its path by updating the corresponding variable in the MUESLI config file. See section 3 (Installation / Papi installation) in the *MUESLI Installation Guide*.

At any place in your program, you must initialize the counter, by the `msFlops` routine, then calling it again to obtain a flops count. Typically, flops counts are used to measure performance of numerical algorithms because they lead to more accurate results than using the CPU time, especially for short computations.

▶ Example #4-69:

```
! integer*8 :: flops
print *, "one FP operation :"
call msFlops(init=0) ! flops initialization
x = 1.0d0 + 1.0d0
call msFlops(count=flops)
print *, "flops = ", flops
```

▷ Output of ex. #4-69:

```
one FP operation :
flops =  1
```

▶ Example #4-70:

```
! integer*8 :: flops
print *, "matrix product 100x100 by 100x100 :"
x = mfOnes(100,100)
y = mfOnes(100,100)
call msFlops(init=0) ! flops initialization
u = mfMul( x, y )
call msFlops(count=flops)
print *, "flops = ", flops
```

▷ Output of ex. #4-70:

```
matrix product 100x100 by 100x100 :
flops =  2000000
```

If you obtain errors in getting flops (typically, all calls to `msFlops` returns the '-1' value), see again the *MUESLI Installation Guide*.

### 4.3.4 Managing Memory

The fundamental derived type used in this library, the `type(mfArray)`, consists actually of several storage arrays declared as pointer[5]. Deallocation of pointer objects is not automatic, so it is under the responsibility of the programmer. Most of time, these deallocations are made by the library itself, however it remains some cases where it must be made inside the user program. For this reason, it is a good practice to deallocate an `mfArray` as soon as it becomes unused.

The deallocation of all internal storage arrays in an `mfArray` is made by using the `msRelease` routine:

▶ Example #4-71:

```
A = mfRand(100,100)
! ... (computation)
call msRelease( A )
```

If you don't use any of these deallocating routine, you could obtain a warning message at the end of your program execution (according your compiler). Further information about this topic can be found in the *MUESLI inside* document.

### 4.3.5 Creating your own `mf-` or `ms-` routines

You can create your own routines to manipulate `mfArray`s but you should be aware of some important things:

- you should firstly remember that MUESLI can automatically deallocate temporary objects, so you must protect the `mfArray` arguments of your routine. This can be realized by using `msInitArgs`. You must also unprotect them at the end of the routine by using the `msFreeArgs` routine.
- secondly, if your routine is a function that returns an `mfArray`, this latter object must be tagged as temporary by `msReturnArray`.

The following example show a function which returns the symmetric part of a matrix.

[5] see the *MUESLI inside* document

► Example #4-72:

```
function sym_part( A ) result( out )
   type(mfArray) :: A
   type(mfArray) :: out
   ! returns the symmetric part of the mfArray 'A'
   integer :: i, j, n
   type(mfArray) :: alpha, beta

   call msInitArgs( A ) ! protects the input argument
!-------------------------------------------------------------------
   ! input matrix must not be empty
   if( mfIsEmpty(A) ) then
      print *, "sym_part: empty matrix."; go to 99
   end if

   ! input matrix must be square
   if( .not. mfIsSquare(A) ) then
      print *, "sym_part: non square matrix."; go to 99
   end if

   n = mfSize(A,1)
   if( n == 1 ) then
      ! trivial case
      out = A
   else
      out = mfZeros(n)
      do i = 1, n
         do j = 1, i-1
            alpha = mfGet(A,i,j)
            beta  = mfGet(A,j,i)
            call msSet( 0.5d0*(alpha+beta), out, i, j )
            call msSet( 0.5d0*(alpha+beta), out, j, i )
         end do
         ! diagonal
         call msSet( mfGet(A,i,i), out, i, i )
      end do
   end if

   call msRelease( alpha, beta )
!-------------------------------------------------------------------
99 continue
   call msFreeArgs( A ) ! unprotects the input argument ...
   call msAutoRelease( A ) ! ... and deallocate it if needed
   call msReturnArray( out ) ! marks the ouput as temporary

end function sym_part
```

### 4.3.6 Numerical precision

All internal computations are done in double precision (see the *IEEE-754* standard).

### 4.3.7 Debugging in MUESLI

The following routines may help in getting information about the internal behavior of the library: `mfGetMsgLevl`, `msSetMsgLevl`, `mfGetTrbLevel`, `msSetTrbLevel`. Information about these routines can be found in the *MUESLI Reference Manual* document. See also the *MUESLI Inside* document.

### Locating your run-time errors...

Because MUESLI uses automatic structures in the `mfArray` derived type, you will face, sooner or later, run-time errors – note that they cannot be detected at compile-time. Though locating these errors is possible by using a debugger (for example, GNU `gdb`), it may lead to problems because each Fortran 90 optional argument has a null address when they are not used, and this sometimes hangs the debugger...

Since the beginning of its design, the library MUESLI shows you the stack of called routines when such errors arise. This is the "backtrace" feature. This facility may even be used by the user himself, by calling the `msMuesliTrace()` routine (see more information in the *MUESLI Reference Manual* document).

### Detecting Floating-Point instabilities

The `msSetRoundingMode` routine may be used to change the rounding mode used by the processor to perform floating-point operations. From a global point of view, it can be judicious to change the rounding mode if you suspect some instabilities in your code. If numerical results change a lot between runs using different rounding modes, you will be able to conclude to a floating-point instability.

However, you will not be able to find the exact location of such floating-point unstabilities with MUESLI: specialized libraries, like *CADNA*[6], are devoted to this kind of difficult problem.

### Detecting *Inf*s and *NaN*s

Nowadays, like default other programs' behaviour, MUESLI operates in full IEEE-754 mode, *i. e.* doesn't stop on floating-point exceptions but produces *Inf*s and *NaN*s (resp. Infinities and Not-a-Number). If you are interested in finding which statement creates these special IEEE values, you must use the following routine: `msEnableFPE`, which is usually called at the beginning of your program.

However, *Inf*s and *NaN*s are sometimes expected (*e. g.* , *NaN*s may represent holes in data whereas *Inf* is the valid and expected result of $\tan(\pi/2)$); so you may want to prevent the exception trapping for certain parts of your code, by using the other routine: `msDisableFPE`.

▶ Example #4-73:

```
!real :: tab(1000)
!type(mfArray) :: x

call msEnableFPE( "usual_exceptions" )

... ! some computation

call msDisableFPE( "usual_exceptions" )
x = 1.0d0/0.0d0 ! no FP exception because trapping is disabled by the
                ! previous statement
call msEnableFPE( "usual_exceptions" )

x = 0.0d0
y = 1/x ! no FP exception because the current division becomes a MUESLI
        ! operation (operator overload), for which FP exceptions are
        ! usually internally disabled.

print *, "using uninitialized data..."
tab(:) = acos(tab(:)) ! FP exception because tab(:) is not initialized and
                      ! is not an mfArray. One can be pretty sure that at
                      ! least one value of tab(:) is out of [-1,1]
```

[6] http://www-anp.lip6.fr/cadna/

▷ Output of ex. #4-73: (when using the GNU Fortran compiler)

```
 using uninitialized data...

Floating point exception: Invalid operation
At line 21 of file test.f90
```

As shown in the previous example, all internal routines are protected against FP exceptions, so it is not always easy to detect programmer's defects. Moreover, you could wonder what is the behaviour of MUESLI when using inverse trigonometric functions in your program: by default, when it is possible, a complex value is returned like in the following example (but this default behaviour can be changed, via the msSetAutoComplex routine):

► Example #4-74:

```
   !type(mfArray) :: x

   x = 1.5d0
   call msDisplay( x, "x" )

   print *, "Auto-Complex conversion = ", mfGetAutoComplex()

   call msDisplay( mfACos(x), "mfACos(x)" )

   call msSetAutoComplex( .false. )
   print *, "Auto-Complex conversion = ", mfGetAutoComplex()

   call msDisplay( mfACos(x), "mfACos(x)" )

   call msEnableFPE( "usual_exceptions" )

   call msDisplay( mfACos(x), "mfACos(x)" )
```

▷ Output of ex. #4-74: (when using the GNU Fortran compiler)

```
 x =

    1.5000

 Auto-Complex conversion =  T

 mfACos(x) =

   0.0000 + 0.9624i

 Auto-Complex conversion =  F

 mfACos(x) =

      NaN

Floating point exception: Invalid operation
At line 159 of file mod_elfun.F90
Traceback: (Innermost first)
   Called from line 19 of file test_AutoComplex.f90
```

### Using specialized options for some solvers

When using the non-linear solvers (mfFSolve and mfLsqNonLin) or differential integrators using implicit schemes (mfDaeSolve on one hand, mfOdeSolve with the BDF method on the other hand), the numerical computation is often much more efficient when the jacobian is provided by the user, and not approximated internally via numerical finite differences. Of course, convergence is related to the quality of the user-supplied jacobian routine, so special attention must be paid when writing the jacobian routine, especially when it is done by hand.

Muesli offers tools for checking the correctness of the jacobian. This can be achieved by using the check_jac option (in the mf_NL_Options and mf_DE_Options derived types).

▶ Example #4-75: Vector function for which we want to find two parameters {p(1),p(2)} such that the sum of the square of all components is minimum.

```
subroutine exp_min( m, n, p, fvec, iflag )
   integer,               intent(in) :: m, n
   real(kind=MF_DOUBLE), intent(in) :: p(n)
   real(kind=MF_DOUBLE)             :: fvec(m)
   integer                          :: iflag
   !---
   integer :: i
   do i = 1, m
      fvec(i) = 2.0d0 + 2.0d0*i - exp(i*p(1)) - exp(i*p(2))
   end do
end subroutine exp_min
```

▶ The jacobian of the previous vector-function is defined here. Note the explicit modification made for one component: we will try to find automatically this error.

```
subroutine exp_min_jac( m, n, p, fjac )
   integer,               intent(in) :: m, n
   real(kind=MF_DOUBLE), intent(in) :: p(n)
   real(kind=MF_DOUBLE)             :: fjac(m,n)
   !---
   integer :: i
   do i = 1, m
      fjac(i,1) = - i*exp(i*p(1))
      fjac(i,2) = - i*exp(i*p(2))
   end do
   !============================================
   fjac(5,1) = fjac(5,1)*1.01d0 ! introduce an explicit error
   !============================================
end subroutine exp_min_jac
```

▶ The minimization is done via the mfLsqNonLin routine, asking the algorithm to do a quick check of the jacobian at each iteration.

```
m = 10
n = 2
p = [ 0.3d0, 0.4d0 ]
call msRelease( options_NL )
options_NL%tol = 1.0d-3
options_NL%check_jac = 1
options_NL%print_check_jac = 1
z = mfLsqNonLin( m, exp_min, p, n, options_NL, jac=exp_min_jac )
```

▷ Output of ex. #4-77:

```
(MUESLI LsqNonLin:) Warning: iteration: 001

quality =

   0.5836
   0.4934
   0.4941
   0.4800
   0.1469
   0.4690
   0.4651
   0.4619
   0.4586
   0.4554

(MUESLI LsqNonLin:) Warning: user jacobian appears as
                    probably *** incorrect ***!
(MUESLI LsqNonLin:) Warning: quick check of the user jacobian:
                    min. of quality vector is: iteration: 009
                            and occurs at row: 5
```

In the previous output, we see that the `LsqNonLin` algorithm detect a strong discrepancy in the fifth equation. We have to do a full check to find the exact location of this error in the jacobian, as shown below.

► Now, we ask `LsqNonLin` routine to do a full check of the jacobian at each iteration.

```
m = 10
n = 2
p = [ 0.3d0, 0.4d0 ]
call msRelease( options_NL )
options_NL%tol = 1.0d-3
options_NL%check_jac = 2
options_NL%print_check_jac = 1
z = mfLsqNonLin( m, exp_min, p, n, options_NL, jac=exp_min_jac )
```

▷ Output of ex. #4-78: With the full check, the exact location of the error is found.

```
(MUESLI LsqNonLin:) Warning: iteration: 001

errors in the jacobian =

   0.0000    0.0000
   0.0000    0.0000
   0.0000    0.0000
   0.0000    0.0000
   0.0099    0.0000
   0.0000    0.0000
   0.0000    0.0000
   0.0000    0.0000
   0.0000    0.0000
   0.0000    0.0000

(MUESLI LsqNonLin:) Warning: user jacobian appears as more or less accurate!
                    (use another option to inspect the err matrix)
(MUESLI LsqNonLin:) Warning: quick check of the user jacobian:
                    min. of quality vector is: iteration: 009
                            and occurs at row: 5  1
```

Be aware that it is not unusual to get strange results for these checks, especially if the system of equations is badly scaled or if floating-point rounding errors occur. The following example deals with such a case.

▶ Example #4-79: Define a DAE system of three equations.

```
!   real(kind=MF_DOUBLE) :: r3_a = -0.5d0, r3_b = 5.0d0, r3_c = -5.0d0

  subroutine resid_3( t, y, yprime, delta, flag )
     real(kind=MF_DOUBLE), intent(in)     :: t, y(*), yprime(*)
     real(kind=MF_DOUBLE), intent(out)    :: delta(*)
     integer,              intent(in out) :: flag
     delta(1) = r3_a*y(1) + r3_b*y(2)*y(3)
     delta(2) = -delta(1) + r3_c*y(2)**2 - yprime(2)
     delta(1) = delta(1) - yprime(1)
     delta(3) = y(1) + y(2) + y(3) - 1.0d0
  end subroutine resid_3

  subroutine jac_resid_3( t, y, yprime, jacobian, cj, nrow )
     real(kind=MF_DOUBLE), intent(in)  :: t, y(*), yprime(*), cj
     integer,              intent(in)  :: nrow
     real(kind=MF_DOUBLE), intent(out) :: jacobian(nrow,*)
     jacobian(1,1) = r3_a - cj
     jacobian(2,1) = -r3_a
     jacobian(3,1) = 1.0d0
     jacobian(1,2) = r3_b*y(3)
     jacobian(2,2) = -r3_b*y(3) + 2*r3_c*y(2) - cj
     jacobian(3,2) = 1.0d0
     jacobian(1,3) = r3_b*y(2)
     jacobian(2,3) = -r3_b*y(2)
     jacobian(3,3) = 1.0d0
  end subroutine jac_resid_3
```

▶ The DAE solver is called as follows:

```
t_span = .t. mf( [ 0.0d0, 0.5d0, 1.0d0, 2.0d0, 5.0d0 ] )
y_0 = [ 1.0d0, 0.0d0, 0.0d0 ]
yp_0 = MF_EMPTY
options_DE%IC_known = .false.
options_DE%check_jac = 1
options_DE%print_check_jac = 1
y = mfDaeSolve( resid_3, t_span, y_0, yp_0, options_DE, jac=jac_resid_3 )
```

▷ Output of ex. #4-80: During integration, the ouput presents many warnings like:

```
(MUESLI DaeSolve:) Warning: time:  2.937688E+00

quality =

   0.4735
   0.4674
   1.0000

(MUESLI DaeSolve:) Warning: user jacobian appears as
                   probably *** incorrect ***!
(MUESLI DaeSolve:) Warning: quick check of the user jacobian:
                   min. of quality vector is: time:  2.937688E+00
                           and occurs at row: 2
```

However, the jacobian is mathematically correct. Moreover, the full check (using `check_jac = 2` in the `options_DE` structure) doesn't detect any error.

If we change the value of the three parameters `[r3_a, r3_b, r3_c]` to their original value (the DAE equations actually describe a stiff chemical system), *i. e.* `[r3_a=-0.04d0, r3_b=1.0d4, r3_c=-3.0d7]`, then both the quick check and the full check claim that the jacobian is inexact:

▷ Output of ex. #4-80: The new DAE system is very stiff (parameters `[r3_a, r3_b, r3_c]` has been changed back to their original values). Case of a quick check which give a wrong result, due to rounding-off errors.

```
(MUESLI DaeSolve:) ERROR: time:  0.000000E+00

quality =

   0.1551
   0.0000
   1.0000

(MUESLI DaeSolve:) ERROR: user jacobian appears as
                   certainly *** wrong ***!
(MUESLI DaeSolve:) ERROR: quick check of the user jacobian:
                   min. of quality vector is: time:  0.000000E+00
                           and occurs at row: 2
```

▷ Output of ex. #4-80: The new DAE system is very stiff (parameters `[r3_a, r3_b, r3_c]` has been changed back to their original values). Case of a full check which give a wrong result, due to rounding-off errors.

```
(MUESLI DaeSolve:) ERROR: time:  0.000000E+00

errors in the jacobian =

   0.0000    0.0000    0.0000
   0.0000    0.4470    0.0000
   0.0000    0.0000    0.0000

(MUESLI DaeSolve:) ERROR: user jacobian appears as pretty *** wrong ***!
                   (max. of relative error is greater than 10 %)
(MUESLI DaeSolve:) ERROR: quick check of the user jacobian:
                   min. of quality vector is: time:  0.000000E+00
                           and occurs at row: 2  2
```

For the DAE solver (`mf/msDaeSolve`), the jacobian must not be singular. The `jac_rcond_min` option of the `mf_DE_Options` derived type allows the user to choose the threshold of the singularity. Moreover, using the `jac_investig` option in the same structure allows the user to know the nullspace of the jacobian when it is singular; this is useful to discover, for example, that two rows of the jacobian are exactly the same (this can come from an error while coding the jacobian or from a badly formed set of DAE equations). See the following example.

▶ Example #4-81: This DAE system contains two identical equations

```
subroutine resid_singular( t, y, yprime, delta, flag )
   real(kind=MF_DOUBLE), intent(in)     :: t, y(*), yprime(*)
   real(kind=MF_DOUBLE), intent(out)    :: delta(*)
   integer,              intent(in out) :: flag
   delta(1) = y(1) + y(3)
   delta(2) = y(2) - yprime(3)
   delta(3) = y(3) - yprime(4)
   ! this equation 4 is exactly the same as equation 2.
   delta(4) = delta(2)
end subroutine resid_singular
```

▶ We solve it using the following calls with the `jac_investig` option

```
t_span = .t. mf( [ 0.0d0, 1.0d0, 2.0d0 ] )
y_0 = [ 0.0d0, 0.0d0, 0.0d0, 0.0d0 ]
options_DE%IC_known = .false. ! Initial Conditions are not known
options_DE%jac_investig = .true.
options_DE%save_sing_jac = .true.
y = mfDaeSolve( resid_singular, t_span, y_0, yp_0, options_DE )
```

▷ Output of ex. #4-81:

```
(MUESLI mfDaeSolve:) Error: the DAE solver encountered the following problem:
                     'Jacobian is singular during the integration phase'

(MUESLI:) processing the Jacobian matrix...

Jacobian matrix has been saved in 'jacobian.dat'.

rational basis of the null space =

    0
   -1
    0
    1

dim(nullspace) =

    1

Above can be found an approximation of the null space. It may help
you detecting wrong equations in your system. Each column is a null
vector of the linear application and it shows, via its components,
the dependences between the current equations coded in the 'resid()'
routine.
(note: the transpose of the jacobian matrix has been processed)

A rational basis of the null space has been saved in 'nullspace.dat'.
```

▷ We can easily verify that the entries of the approximated jacobian matrix, saved in the file `jacobian.dat`, are all correct with respect to the differential equations in the `resid_singular` routine:

```
1.00E+000  0.00E+000  1.00E+000  0.00E+000
0.00E+000  1.00E+000 -4.00E+003  0.00E+000
0.00E+000  0.00E+000  1.00E+000 -4.00E+003
0.00E+000  1.00E+000 -4.00E+003  0.00E+000
```

Coming back to the output of `mfDaeSolve` which contains the main components of the null space, it clearly shows that equations 2 and 4 are the same.

Note that this investigation is done only in the DEBUG mode and that it may be very expensive, according to the size of the jacobian matrix.

### 4.3.8 Exchanging data with MATLAB

MUESLI provides a way to build MATLAB mex functions. These executables (seen as new commands under MATLAB), `mbfread` and `mbfwrite`, are used to read and write MBF files, *i. e.* binary files created by the `msSave` routine (cf. section 4.2.8), in gzipped/not gzipped state, in little/big endian format, of real/complex type and for sparse structure as well as dense structure.

The use of these MATLAB commands are not described in the *MUESLI Reference Manual*. However, you can type '`help mbfread`' and '`help mbfwrite`' under MATLAB to get a short usage description.

Matrix properties (nor physical properties) are not read in MATLAB. On the other hand, the `mbfwrite` mex function writes a valid MBF file, always setting UNKNOWN to the two matrix properties and always returning a dimensionless `mfArray`.

Please see the *MUESLI Installation Guide* document for building these mex functions.

## 4.4 Equivalence between MUESLI routines, MATLAB commands and f90 intrinsic functions

The usual mathematical functions (sin, cos, ..., log, exp, sqrt, ...) are not included in the following tables. Note that in MUESLI the following functions: acos, asin, atan, cosh, sinh, tan, tanh have been extended to support full range of real and complex values, including special IEEE-754 values (*Inf* and *NaN*).

### 4.4.1 from MUESLI routines

| MUESLI | MATLAB | Fortran 90 |
|---|---|---|
| MF_EPS | eps | epsilon |
| MF_REALMAX | realmax | huge |
| MF_REALMIN | realmin | tiny |
| operator(.t.) | ' | transpose |
| mfAll | all | all |
| mfAny | any | any |
| mfCeil | ceil | ceiling |
| mfComplex | complex | complex |
| mfConj | conj | conjg |
| mfCount | nnz (1) | count |
| mfCshift | – | cshift |
| mfEoshift | – | eoshift |
| mfFix | fix | – |
| mfFloor | floor | floor |
| mfImag | imag | aimag |
| mfMul | * | matmul |
| mfMax | max | maxval |
| mfMerge | – | merge |
| mfMin | min | minval |
| mfPack | – | pack |
| mfProd | prod | product |
| mfRand | rand | random_number |
| mfRound | round | anint |
| mfReal | real | real |
| mfReshape | reshape | reshape |
| mfShape | size | shape |
| mfSize | prod and size (2) | size |
| mfRepMat | repmat | spread |
| mfSum | sum | sum |
| mfUnpack | – | unpack |

*notes:*
(1) nnz operates globally on the array whereas count and mfCount may operate by columns.
(2) under MATLAB, the total size of an array A (*i. e.* the number of its elements) writes prod(size(A)).

### 4.4.2 from MATLAB commands

| MATLAB | MUESLI | Fortran 90 |
|---|---|---|
| * | mfMul | matmul |
| ' | operator(.t.) | transpose |
| all | mfAll | all |
| any | mfAny | any |
| ceil | mfCeil | ceiling |
| complex | mfComplex | complex |
| conj | mfConj | conjg |
| eps | MF_EPS | epsilon |
| fix | mfFix | – |
| floor | mfFloor | floor |
| imag | mfImag | aimag |
| max | mfMax | maxval |
| min | mfMin | minval |
| nnz | mfNnz | count and size |
| prod | mfProd | product |
| rand | mfRand | random_number |
| round | mfRound | anint |
| real | mfReal | real |
| realmax | MF_REALMAX | huge |
| realmin | MF_REALMIN | tiny |
| reshape | mfReshape | reshape |
| size | mfShape | shape |
| repmat | mfRepMat | spread |
| sum | mfSum | sum |

### 4.4.3 from f90 intrinsic functions

| Fortran 90 | MUESLI | MATLAB |
|---|---|---|
| aimag | mfImag | imag |
| anint | mfRound | round |
| all | mfAll | all |
| any | mfAny | any |
| ceiling | mfCeil | ceil |
| complex | mfComplex | complex |
| conjg | mfConj | conj |
| count | mfCount | nnz (1) |
| cshift | mfCshift | – |
| dot_product | mfMul (3) | * and ' (3) |
| eoshift | mfEoshift | – |
| epsilon | MF_EPS | eps |
| floor | mfFloor | floor |
| huge | MF_REALMAX | realmax |
| matmul | mfMul | * |
| maxloc | mfMax (4) | max (4) |
| maxval | mfMax | max |
| merge | mfMerge | – |
| minloc | mfMin (4) | min (4) |
| minval | mfMin | min |
| pack | mfPack | – |
| product | mfProd | prod |
| random_number | mfRand | rand |
| random_seed | mfRand (4) | rand (4) |
| real | mfReal | real |
| reshape | mfReshape | reshape |
| shape | mfShape | size |
| size | mfSize | prod and size (2) |
| spread | mfRepMat | repmat |
| sum | mfSum | sum |
| tiny | MF_REALMIN | realmin |
| transpose | operator(.t.) | ' |
| unpack | mfUnpack | – |

*notes:*
(1) `nnz` operates globally on the array whereas `count` and `mfCount` may operate by columns.
(2) under MATLAB, the total size of an array `A` (*i. e.* the number of its elements) writes `prod(size(A))`.
(3) if `a` and `b` are column vectors, then `dot_product(a,b)` writes `mfMul(.t.a,b)` with MUESLI and `a'*b` under MATLAB.
(4) an additional argument must be used.

# 5 FGL (Graphical Library)

FGL is a library which depends on FML. Most of FGL routines work with `mfArrays`. Usually, the two forms of the same routine (*i. e.* , the subroutine, beginning with '`ms`', and the function, beginning with '`mf`') do the same job, but the latter returns an identifier (most of time a graphic handle).

It is recommended to use the routine `msExitFgl` at the end of the graphic part of your program; indeed, not only this routine waits for a user answer – avoiding the close of all opened windows – but it also cleans the graphic internal memory.

## 5.1 Creating and Numbering Figures

A simple call to `msFigure` opens a window in which all the following graphic commands will be displayed. Two optional arguments let you to set (i) the number of this figure (in the case where you have multiple figures) and (ii) the position and the size of the window.

Even if it is not specified at the creation, each figure (*i. e.* each window) has a number which can be later retrieved by the function `mfFigure`. This number is required to make active or to close a window.

`msInitFgl` must be called before opening any window if you want to have the name of the executable program displayed in the window title.

▶ Example #5-0:

```
call msInitFgl() ! gets the name of the executable
call msFigure(5) ! opens a window for the FGL figure #5
! ... next graphic commands are output in FGL figure #5
position = [ 100, 100, 800, 600 ] ! prepares for next figure
call msFigure(3,position) ! opens a window for the FGL figure #3, with
                          ! position = (100,100) and size = 800 by 600 pixels
! ... next graphic commands are output in FGL figure #3
call msFigure(5)
! ... next graphic commands are output in FGL figure #5
call msClose(5)

! hereafter, only figure #3 is opened
call msExitFgl() ! figure #3 is visible until the answer 'y'
                 ! is typed by the user
```

By default, the background color of graphic windows is black. However, the user can change it to the white color, by using the `msSetBackgroundColor` routine. A white background color is indeed more consistent with the EPS files printed from the figures.

## 5.2 High-level Plotting Routines

Some routines make simple plots, constitued by linear segments. They are `msPlot`, `msSemilogx`, `msSemilogy`, and `msLoglog`. All these four routines plot a vector `mfArray` against another vector `mfArray` and have the same interface, similar to the '`plot`' Matlab command.

▶ Example #5-1: (see output plot in figure 1)

```
call msSetBackgroundColor( "white" ) ! default would give a black background

call msFigure(1)

x = mfLinSpace( 0.0d0, 1.0d0, 8 )
y = [ 0., 1., 0., 1., 0., 1., 0., 1. ]

call msPlot( x, y )
```

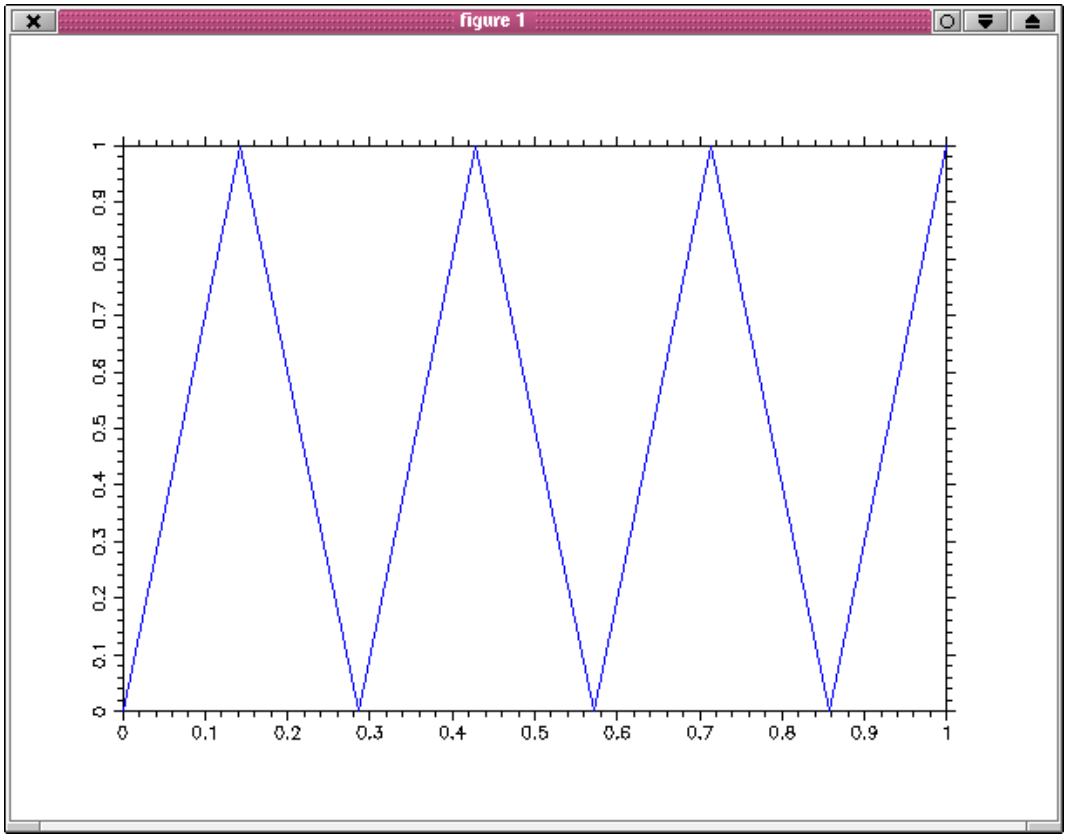



Figure 1: output of ex. #5-1: msPlot use

While the previous plot routines draw linear segment between data points, two additional routines are available to draw curved lines. The first one, `msPlotCubicBezier` takes the data vector as control points for a piecewise cubic Bézier curve.

`msHold` is used to plot a new graphic object without erasing the current figure.

▶ Example #5-2: (see output plot in figure 2)

```
! integer :: n_seg, i, ioff

call msFigure(1)

call msAxis( [ -0.1d0, 1.1d0, -0.1d0, 1.1d0 ] )
call msAxis( "equal" )

x = [ 0., 0.3333, 0.3333, 0.6667, 0.6667, 1., 1. ]
y = [ 0., 0., 1., 1., 0., 0., 1. ]

call msPlotCubicBezier( x, y ) ! default color is blue

! control points drawing (in red)
call msHold( "on" )
call msPlot( x, y, "r+" )
n_seg = (Size(x)-1)/3
do i = 1, n_seg
   ioff = 3*(i-1)
   call msPlot( mfGet(x,ioff+1 .to. ioff+2),                    &
                mfGet(y,ioff+1 .to. ioff+2), "r--" )
   call msPlot( mfGet(x,ioff+3 .to. ioff+4),                    &
                mfGet(y,ioff+3 .to. ioff+4), "r--" )
end do
```

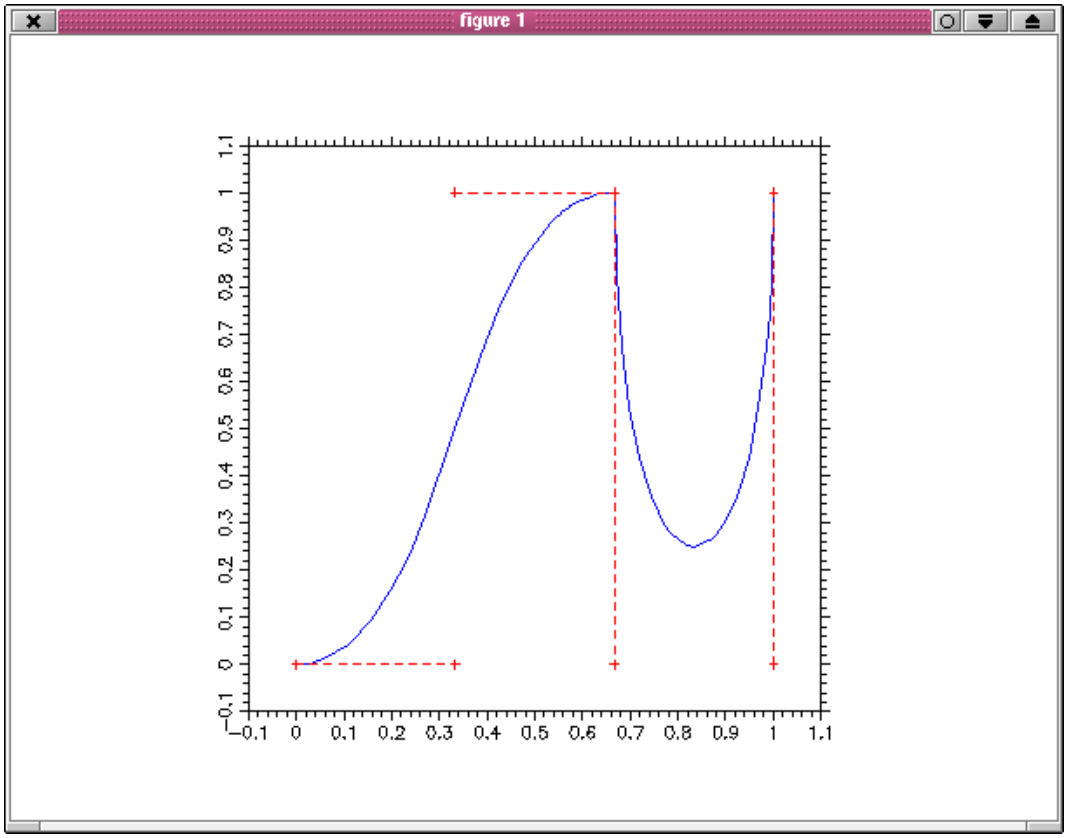



Figure 2: output of ex. #5-2: msPlotCubicBezier use

The second one, `msPlotCubicSpline` interpolates through the data points, and draw a piecewise parametrized cubic spline curve. However, some additional data must have been previously computed

by using the routine `mfSpline`.

▶ Example #5-3: (see output plot in figure 3)

```
call msFigure(1)

abs_curv = [ 0., 1., 2., 3., 4., 5., 6. ]
x = [ 0., 0.3333, 0.3333, 0.6667, 0.6667, 1., 1. ]
y = [ 0., 0., 1., 1., 0., 0., 1. ]

wx = mfSpline( abs_curv, x )
wy = mfSpline( abs_curv, y )

call msPlotCubicSpline( abs_curv, x, y, wx, wy )
call msHold( "on" )
call msPlot( x, y, "ro" ) ! 'o' symbols show the data points
```

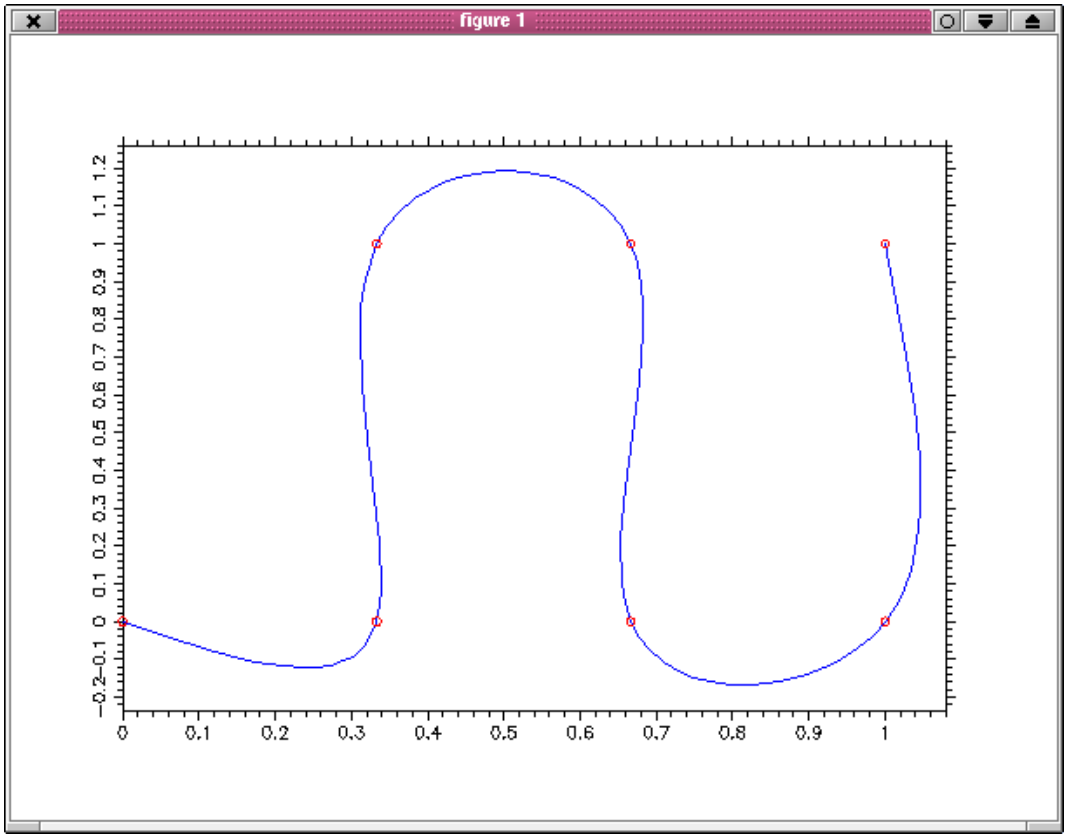



Figure 3: output of ex. #5-3: msPlotCubicSpline use

Two routines work with 2D data. `msContour` draws contours of data contained in an `mfArray`, whereas `msPColor` draws a pseudo-color plot of the same data. Be aware that a colormap must be defined before.

▶ Example #5-4: (see output plot in figure 4)

```
call msFigure(1)

! good pratice: not to exceed 256 colors
call msColormapRange( 16, 255 )
call msColormap( "rainbow" )
call msAxis( "equal" )

call msMeshGrid( mfOut(x,y), mfLinspace(0.0d0,  5.0d0, 20),          &
                             mfLinspace(0.0d0, 10.0d0, 40) )
r = mfSqrt(x**2 + y **2)
z = mfSin( r ) / r
call msSet( 1.0d0, z, 1, 1 ) ! avoids NaN value : sin(0)/0 = 1

call msCAxis( [ -0.3d0, +1.0d0 ] ) ! defines the color axis

call msContour( mfGet(x,1,MF_COLON), mfGet(y,MF_COLON,1), z )
call msColorbar( "on" )
```

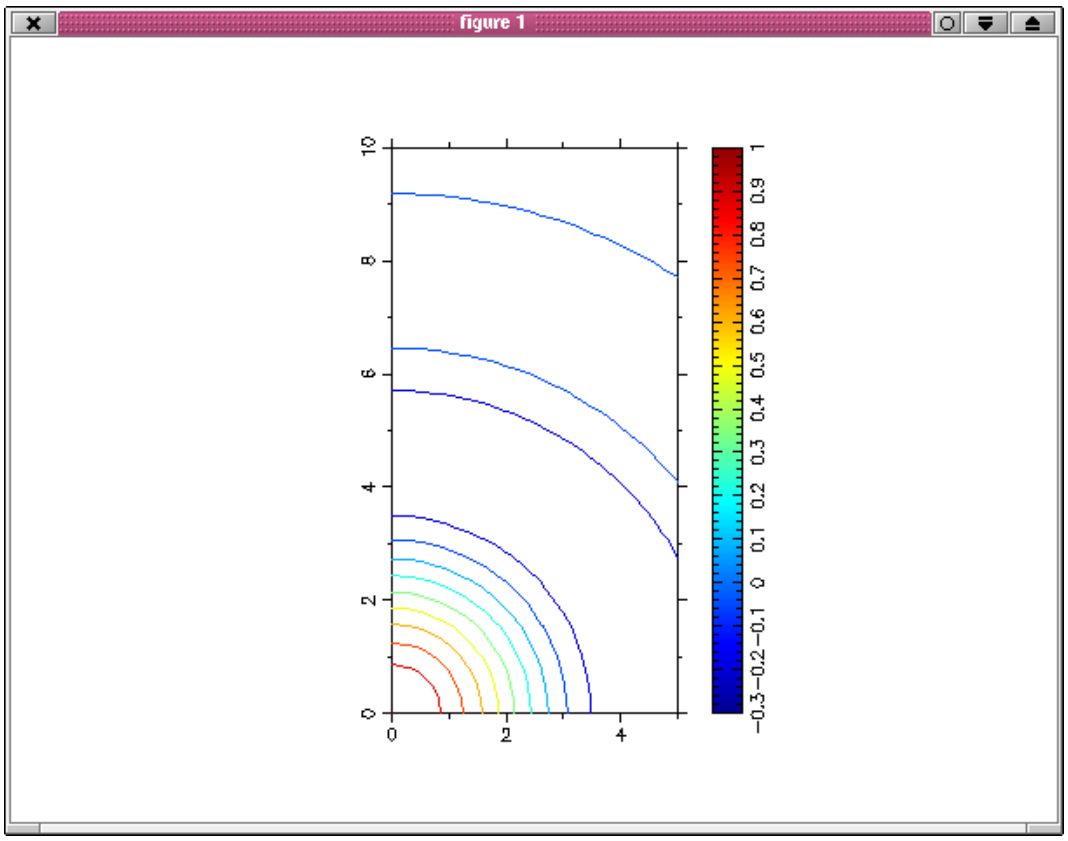



Figure 4: output of ex. #5-4: msContour use

▶ Example #5-5: (see output plot in figure 5)

```
call msFigure(1)

call msColormapRange( 16, 255 ) ! 256 shades
call msColormap( "rainbow" )
call msAxis( "equal" )

x = mfRandN( 10 )
call msCAxis( [ -3.0d0, +3.0d0 ] )

call msPcolor( x ) ! default: shading is "interp"

call msColorbar( "on" )
```

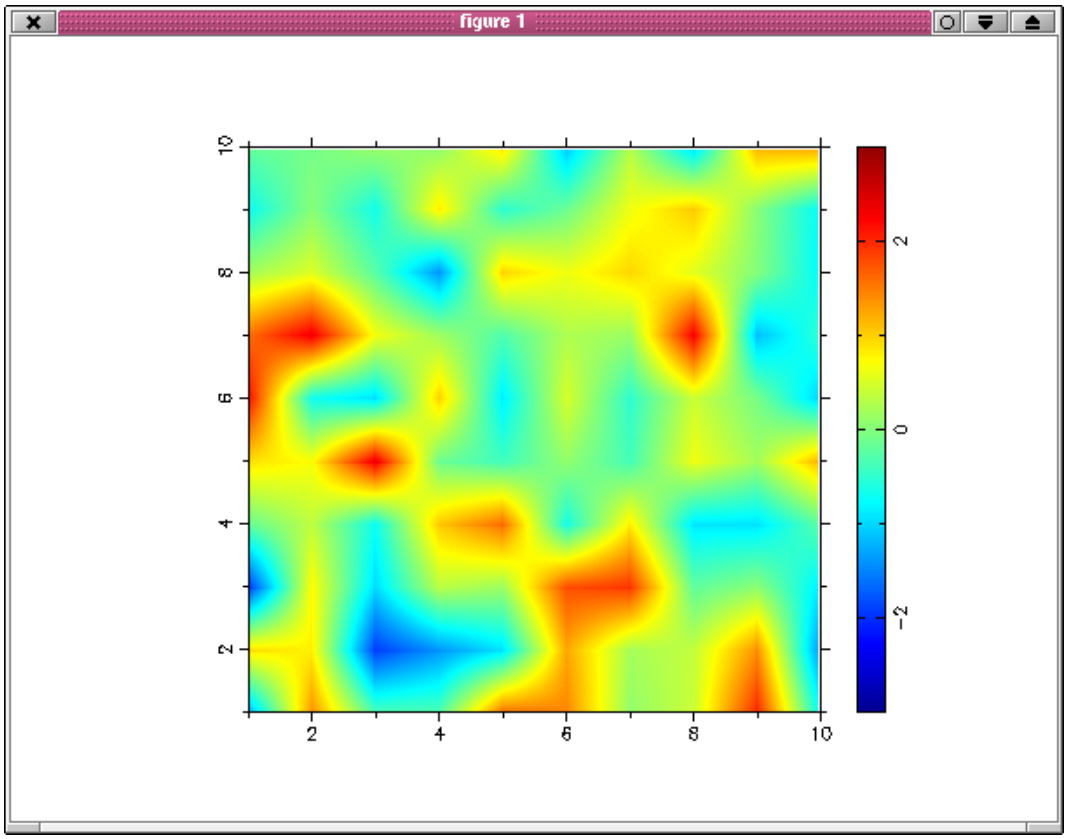



Figure 5: output of ex. #5-5: msPColor use (default shading)

Note that for `mfArrays` with great size (*i. e.* giving small cells) the 'flat' shading is sufficient and much more efficient.

▶ Example #5-6: (see output plot in figure 6)

```
call msFigure(1)

call msColormapRange( 16, 255 )
call msColormap( "rainbow" )
call msAxis( "equal" )

x = mfRandN( 10 )
call msCAxis( [ -3.0d0, +3.0d0 ] )

call msShading( "flat" )
call msPcolor( x )

call msColorbar( "on" )
```

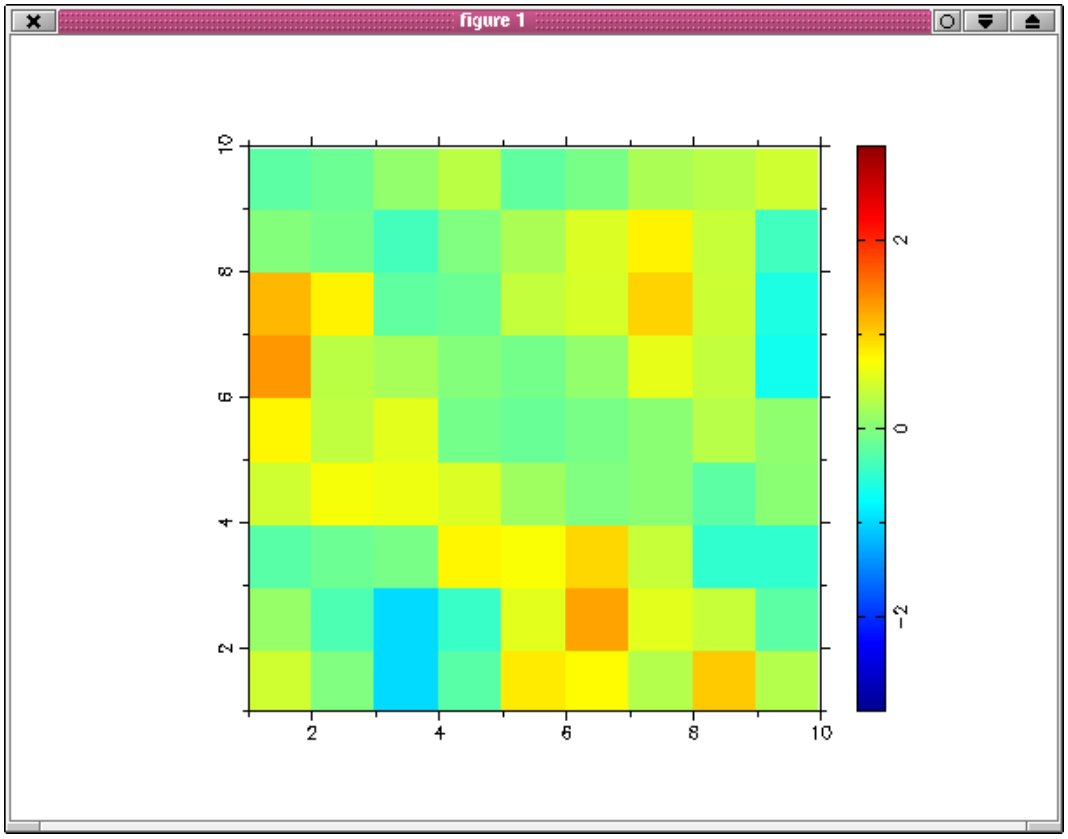



Figure 6: output of ex. #5-6: msPColor use (flat shading)

The two previous routines work only for 2D regular data. Other routines are available when data are spread on an irregular 2D grid.

▶ Example #5-7: (see output plot in figure 7)

```
call msFigure(1)

call msColormapRange( 16, 255 )
call msColormap( "rainbow" )
call msAxis( "equal" )

x = [ 0.05d0, 1.05d0, 1.0d0, -0.05d0, 0.5d0 ]
y = [ 0.0d0, -0.05d0, 1.0d0, 0.95d0, 0.5d0 ]
z = x + y
call msDisplay( .t.(x .vc. y .vc. z), "nodes (x, y, value)" )

tri = mfDelaunay( x, y )
call msDisplay( tri, "triangles" )

call msAxis( [ -0.1d0, 1.1d0, -0.1d0, 1.1d0 ] )
call msCAxis( [ 0.0d0, 2.0d0 ] )
call msTriContour( x, y, z, tri )
call msColorbar( "on" )

! code not shown : plotting the triangles in grey ...
```

▷ Output of ex. #5-7:

```
nodes (x, y, value) =

   0.0500    0.0000    0.0500
   1.0500   -0.0500    1.0000
   1.0000    1.0000    2.0000
  -0.0500    0.9500    0.9000
   0.5000    0.5000    1.0000

triangles =

    4     1     5
    5     2     3
    2     5     1
    5     3     4
```

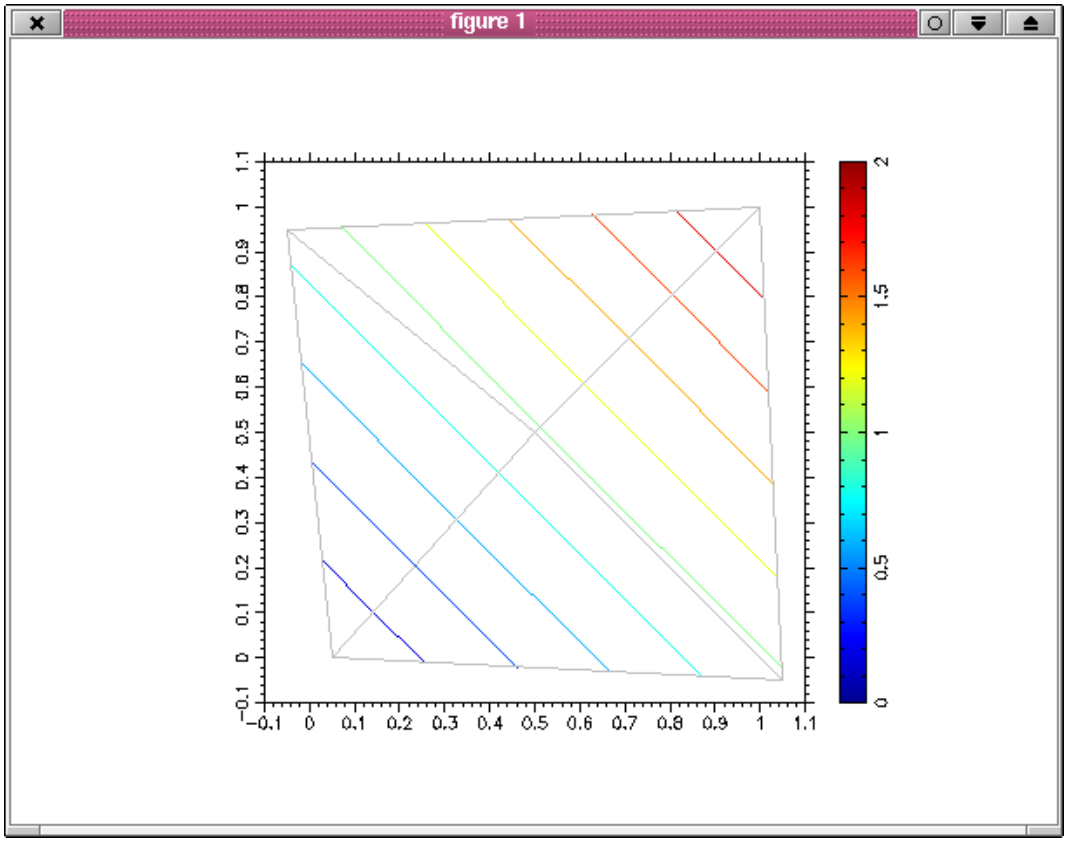



Figure 7: output of ex. #5-7: msTriContour use

▶ Example #5-8: (see output plot in figure 8)

```
call msFigure(1)

call msColormapRange( 16, 255 )
call msColormap( "rainbow" )
call msAxis( "equal" )

x = [ 0.05d0, 1.05d0, 1.0d0, -0.05d0, 0.5d0 ]
y = [ 0.0d0, -0.05d0, 1.0d0, 0.95d0, 0.5d0 ]
z = x + y
call msDisplay( .t.(x .vc. y .vc. z), "nodes (x, y, value)" )

tri = mfDelaunay( x, y )
call msDisplay( tri, "triangles" )

call msAxis( [ -0.1d0, 1.1d0, -0.1d0, 1.1d0 ] )
call msCAxis( [ 0.0d0, 2.0d0 ] )
call msTriPcolor( x, y, z, tri )
call msColorbar( "on" )

! code not shown : plotting the triangles in grey ...
```

▷ Output of ex. #5-8:

```
nodes (x, y, value) =

   0.0500    0.0000    0.0500
   1.0500   -0.0500    1.0000
   1.0000    1.0000    2.0000
  -0.0500    0.9500    0.9000
   0.5000    0.5000    1.0000

triangles =

    4     1     5
    5     2     3
    2     5     1
    5     3     4
```

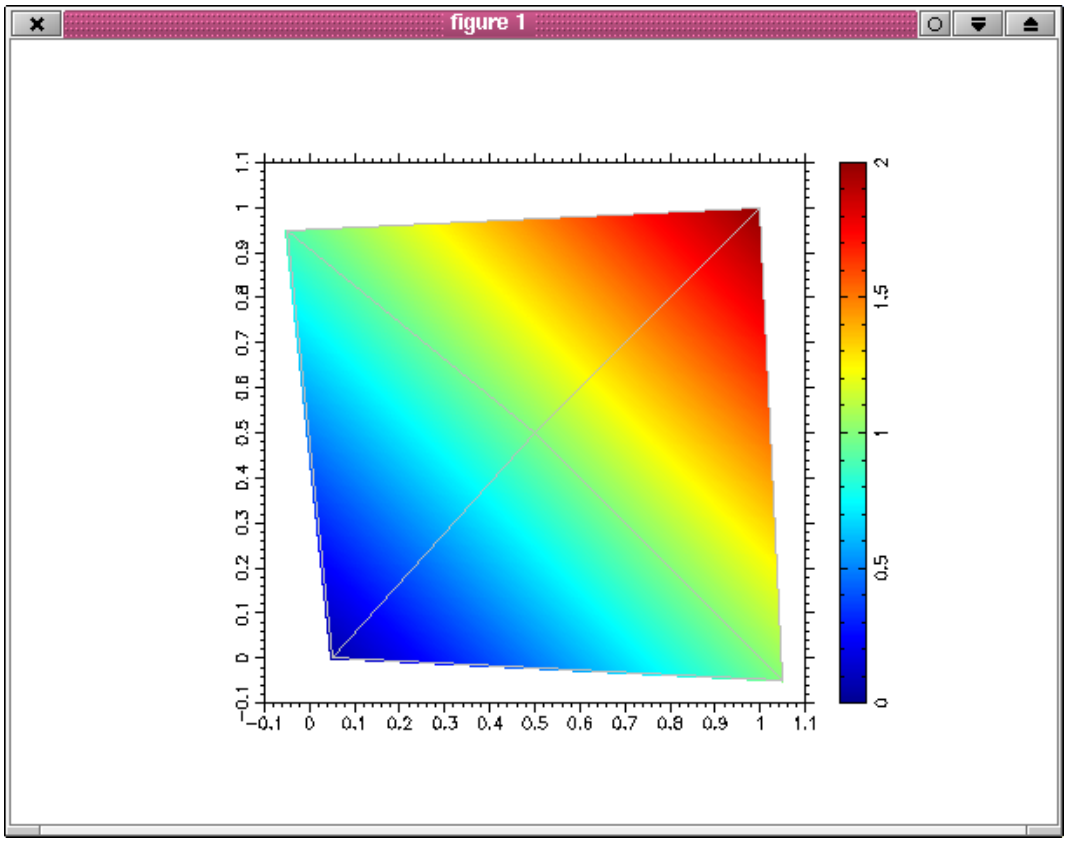



Figure 8: output of ex. #5-8: msTriPColor use

In order to display bitmap images with `msImage`, you must first load the image by using `mfImRead`.

▶ Example #5-9: (see output plot in figure 9)

```
! character(len=80) :: image_name

call msFigure(1)
call msAxis( "equal" )

image_name = "images/XPM_image.xpm"
call msImRead( mfOut(image,cmap), trim(image_name) )
call msColormap( cmap )

call msImage( image )
call msTitle( trim(image_name) )
```

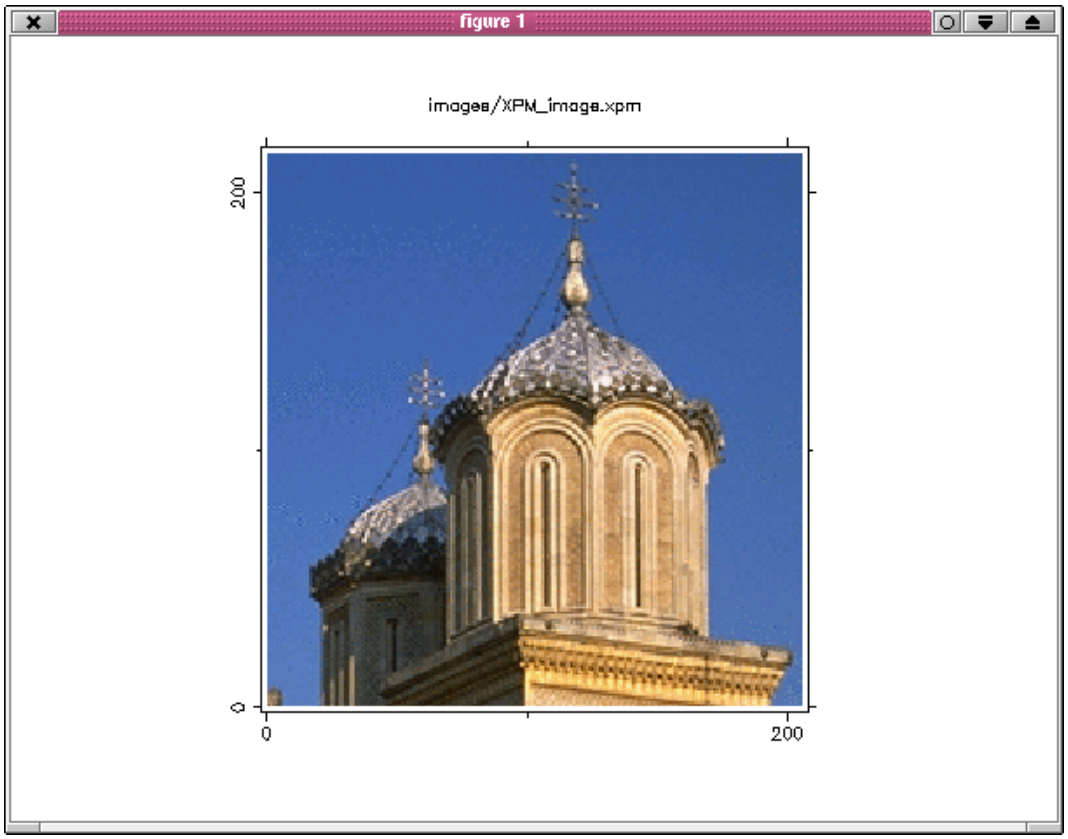



Figure 9: output of ex. #5-9: msImage use

Transparency can be set in your image by two ways:

1. for the real valued pixels storing scheme, set a color (in the colormap `cmap`) to the triplet (`NaN`, `NaN`, `NaN`).
2. for the indexed pixels storing scheme, set an element of the mfArray `image` to `NaN`.

`msPlotHist` can be used to build and plot histograms.

▶ Example #5-10: (see output plot in figure 10)

```
! integer :: n_max, n, nbin
! real(kind=MF_DOUBLE) :: hope

n_max = 1.0e7
n = 1.0e6
nbin = 60
hope = dble(n)/dble(nbin)

call msFigure(2)
call msAxis( [ -3.02d0, 3.02d0, 0.0d0, 0.05d0*n ] )

x = mfRandN( n, 1 )
call msPlotHist( mfOut(), x, -3.0d0, 3.0d0, nbin )

call msHold( "on" )

x = mfLinspace( -3.02d0, 3.02d0, 500 )
y = 0.10*n/sqrt(2.0d0*MF_PI)*mfExp(-0.5d0*x**2)
call msPlot( x, y, "--r" )
call msTitle( "Normal Distribution" )
```

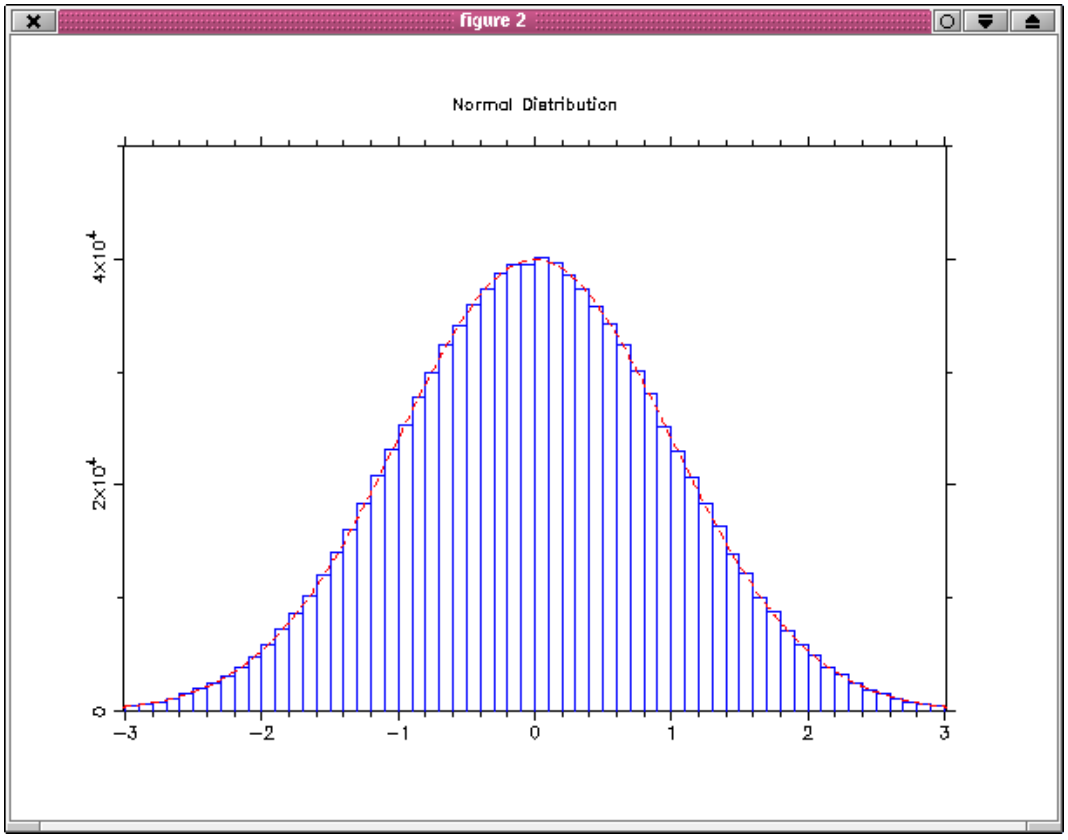



Figure 10: output of ex. #5-10: msPlotHist use

msSpy is exclusively used to plot the pattern of a sparse `mfArray`.

▶ Example #5-11: (see output plot in figure 11)

```
A = mfLoadSparse( "data/C.csc", format="CSC" )
call msSpy( A )
call msTitle("Sparse matrix for a Poisson equation")
```

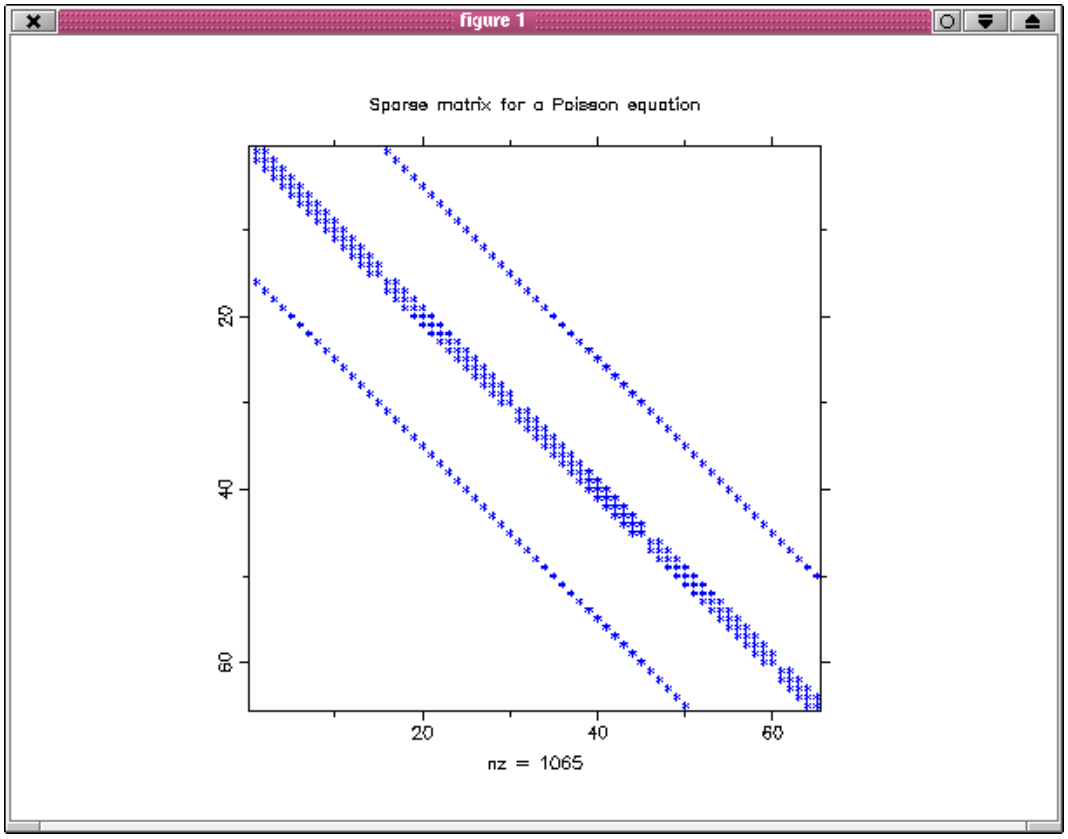



Figure 11: output of ex. #5-11: msSpy use (zoom)

Finally, FGL can also draw a 3D surface from 2D `mfArray` data. Use the `msSurf` routine. When printing a 3D surface plot, you can increase the resolution by choosing a larger window size; you are actually able to choose a very large window size (even more larger than your screen) by using the `msSetX11Device` routine.

▶ Example #5-12: (see output plot in figure 12)

```
call msFigure(1)

x_gen = mfLinspace(-15.0d0, 15.0d0, 50)
y_gen = x_gen

call msMeshgrid( mfOut(x,y), x_gen, y_gen )

r = mfSqrt(x**2 + y **2)
z = mfSin( r ) / r

call msSurf( x_gen, y_gen, z )
call msTitle( "Surface : z = sin(r)/r, with r = sqrt(x\u2\d+y\u2\d)" )
! as specified in the Reference Manual, title and labels must be
! followed by an explicit call to 'msDrawNow()'
call msDrawNow()
```

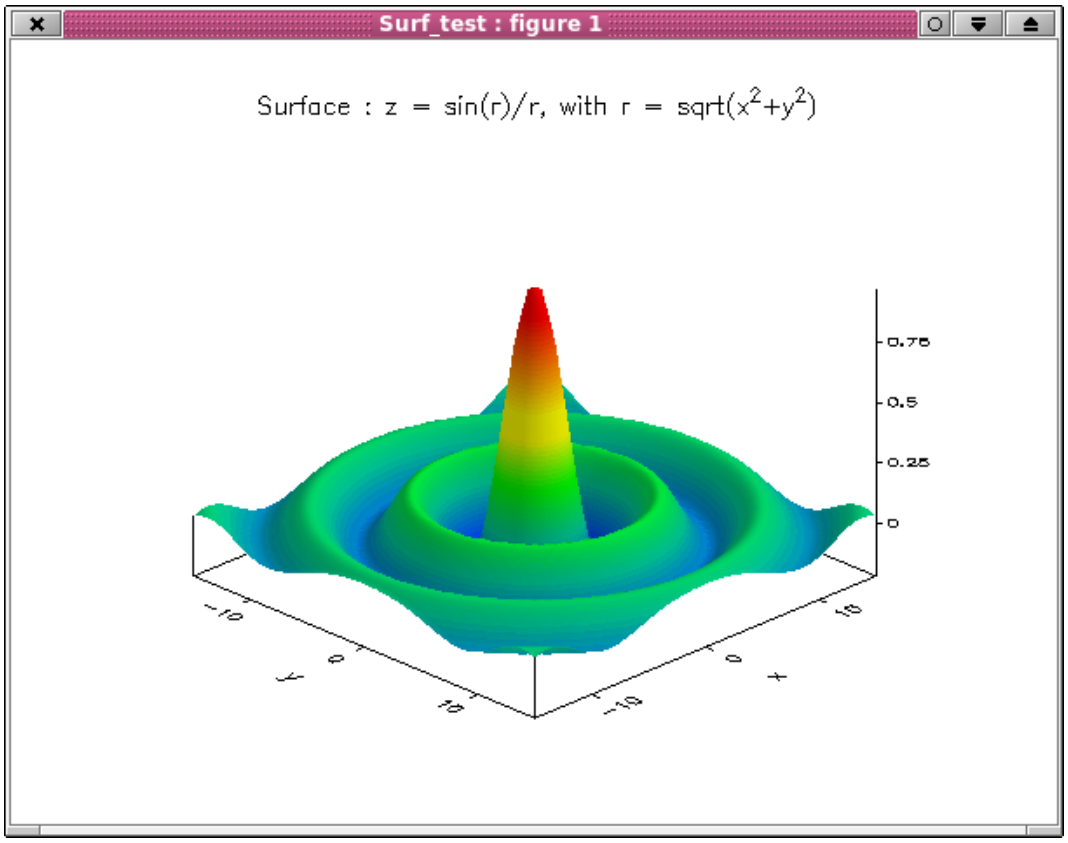



Figure 12: output of ex. #5-12: msSurf use

## 5.3 Interactive Routines

Interactive routines are those which require the user intervention:

- `msPan` translates the axes of the current figure;
- `msZoom` changes the axes of the current figure;
- `msPanAndZoom` does both scrolling and zooming in the current figure;

- `msMoveLegend` moves the legends frame inside the axes;
- `mfGinput` returns in an `mfArray` the coordinates of the clicked point; optionally, you can get which mouse button has been used, or which keyboard key has been pressed;
- `mfBand` returns in an `mfArray` the bounding box of the drawn rectangle;
- `msCustomMove` allows the user to do some action according to the move of the mouse.

In a standard usage, the mouse is used to move the pointer and its five buttons may be detected. When the mouse is not present, the user can use some letters of the keyboard to emulate the mouse:

- capital **L** for the left button;
- capital **M** for the middle button;
- capital **R** for the right button;
- **2** for scroll up (with the wheel);
- **8** for scroll down (with the wheel).

Moreover, be aware that a click of the mouse leads to two different X11 events: a "*MouseDown event*" followed by a "*MouseUp event*". Knowing that, you will have to type two keys to simulate a click: for example, a left-button click is emulated by typing **L** followed by any key. On the contrary, the mouse wheel generates only one X11 event.

The `msCustomMove` routine is, without any doubt, the most complex interactive routine; however it allows powerful graphical programs. Hereafter can be found some comments about the example program `CustomMove_test`, located in the `src/tests/fgl` directory in the *MUESLI* sources.

First the three user routines, arguments of `msCustomMove` (see the *Muesli Reference Manual*) are placed in the same module named `mod_CustomMove`. This module defines some global variables used to communicate with the main program:

```
integer :: h
real(kind=MF_DOUBLE) :: x_text, y_text
real(kind=MF_DOUBLE) :: xref, yref
real(kind=MF_DOUBLE) :: x_center_text, y_center_text, dx_text2
```

For example, `x_center_text` and `y_center_text` are initialized in the main program to the position of the center of the box containing the "Hello" string to be moved.

The first routine:

```
subroutine mouse_down( x, y )
   real(kind=MF_DOUBLE), intent(in) :: x, y
   xref = x
   yref = y
end subroutine mouse_down
```

does nothing special apart saving the initial position of the pointer when the mouse is down.

The second routine:

```
subroutine move( x, y, redraw )
   real(kind=MF_DOUBLE), intent(in) :: x, y
   logical             , intent(in) :: redraw
   ! move only if the click location was close enough to the string
   if( (xref-x_center_text)**2 + (yref-y_center_text)**2 <= dx_text2 ) then
      ! move the grobj via msGSet
      dx = x - xref
      dy = y - yref
      call msGSet( 1, h, "coordinates", [x_text+dx, y_text+dy] )
      x_text = x_text + dx
      y_text = y_text + dy
      x_center_text = x_center_text + dx
      y_center_text = y_center_text + dy
      ! redraw only if required
      if( redraw ) then
         call msDrawNow()
      end if
      xref = x
      yref = y
   end if
end subroutine move
```

is continuously called when the mouse is down. It moves the graphic objet of handle `h` (the "Hello" string) of a distance equal to the displacement of the pointer from its previous location. Note that the redraw of the whole figure (which can contain a great number of graphic objects) is done only if the boolean `redraw` is *TRUE*.
The third routine:

```
subroutine mouse_up( )
end subroutine mouse_up
```

does nothing because in this example, no special action has to be applied at the end of the click.

## 5.4 Annotating a Figure

You may add some additional decorations to a figure, with the use of `msText`, `msXLabel`, `msYLabel` and `msTitle`. Here is an example.

▶ Example #5-13: (see output plot in figure 13)

```
! character(len=72) :: string

call msFigure(1)

! before annotate, the axes must exist
call msAxis( [ -0.05d0, 1.05d0, -0.05d0, 1.05d0] )

string = "color = 'indian red', inclined at 30 degrees"
call msText( 0.0d0, 0.0d0, trim(string), angle=30.0d0,              &
             height=2.0d0, col_name="indian red" )
call msHold( "on" )
call msText( 0.1d0, 0.85d0, "RGB=[0.2,0.3,0.7]", angle=-20.0d0,    &
             height=2.0d0, col_rgb=[0.2d0,0.3d0,0.7d0] )

call msTitle( "The title" )
call msXLabel( "X-axis" )
call msYLabel( "Y-axis" )
```

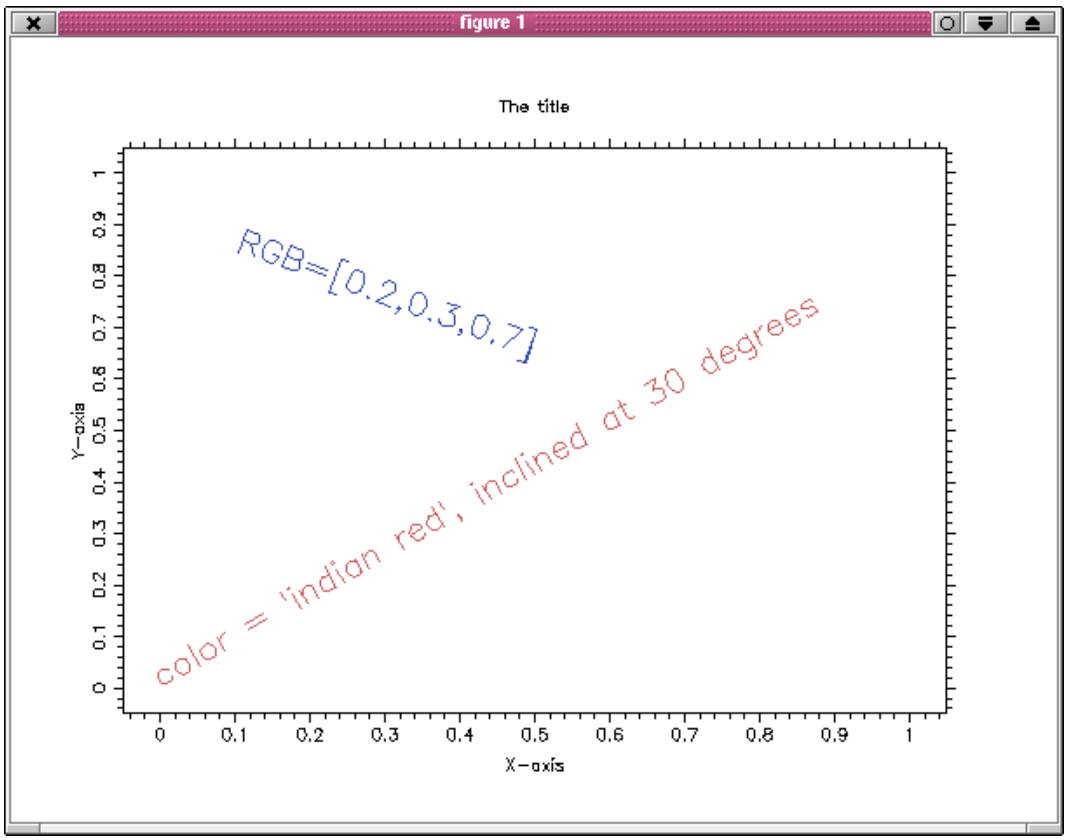



Figure 13: output of ex. #5-13: annotations on a figure

If you have multiple curves plotted in the same figure, you can also use the `msLegend` to add some legend to the curves; up to six curves are supported. An example can be found below.

▶ Example #5-14: (see output plot in figure 14)

```
call msFigure(1)
call msAxis( [ -0.1d0, 1.1d0, -0.1d0, 1.1d0 ] )
call msHold( "on" )

x = [  0., 0.333, 0.333, 0.666, 0.666, 1. ]
y = [  0., 0.,    0.6,    0.6,    0.1,    0.1 ]
call msPlot( x, y )

x = [  0., 0.25, 0.25, 0.5, 0.5, 1. ]
y = [  0.5, 0.5,  0.1,  0.1,   0.25,   0.25 ]
call msPlot( x, y, "r--" )

x = [  0., 0.15, 0.15, 0.75, 0.75, 1. ]
y = [  0.3, 0.3,  0.2,  0.2,   0.4,   0.4 ]
call msPlot( x, y, "mo-" )

x = mfLinSpace( 0.0d0, 1.0d0, 25 )
y = 0.333d0 + 0.333d0*mfSin( 2*MF_PI*x )
call msPlot( x, y, "k+" )

call msText( 0.05d0, 0.05d0, "Hello", height=2.0d0 )

call msLegend( "blue polyline", "red dashed line",                    &
               "magenta marked line", "black crossed sine" )
```

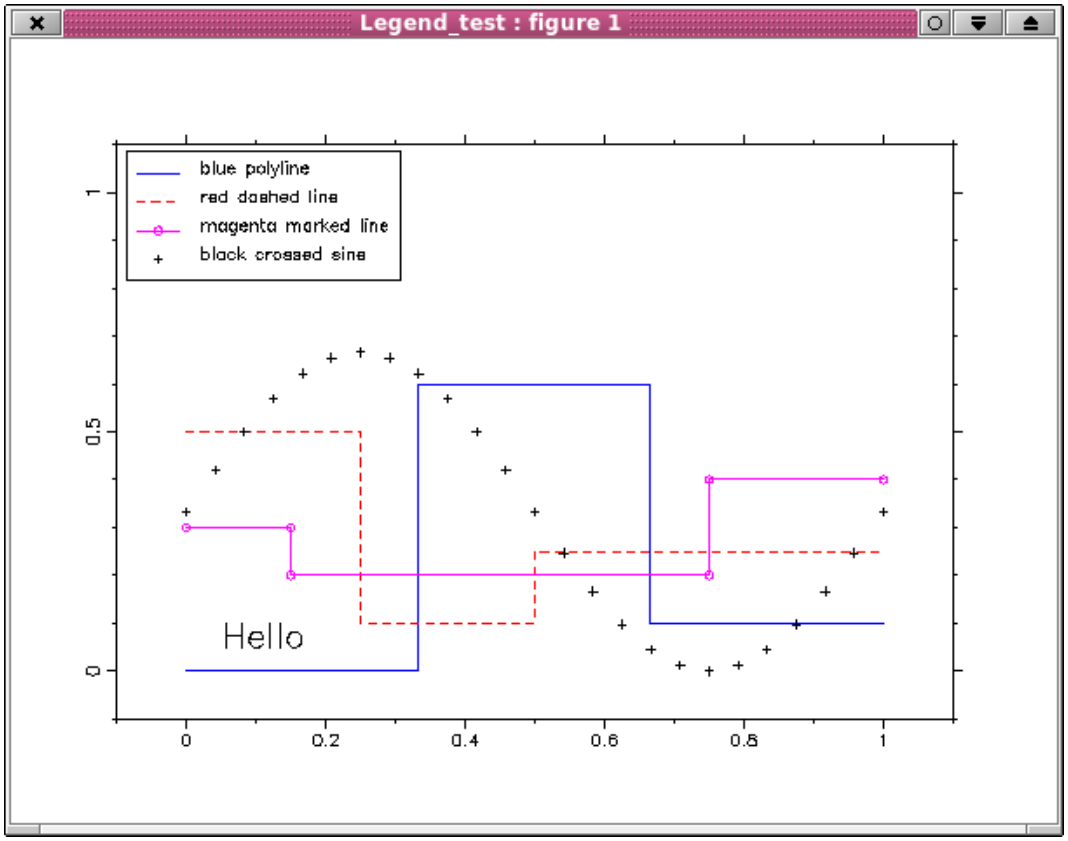



Figure 14: output of ex. #5-14: displaying legends to curves

Once drawn on the figure, this legends frame can be interactively moved via the msMoveLegend. By default, the legends frame is drawn near the top left corner.

Finally, msClf clears the current figure but doesn't close its window.

## 5.5 Text, Symbols and Fonts

In this section, we are describing how characters can be displayed in a figure. When plotted on the screen, characters are digitized by using small straight segments. However, when printing in an EPS file, it is possible to use PostScript Type 1 fonts for a better result, by calling the msSetPSFonts routine.

Letter's and symbol's display is essentially done by the following routines, described in the previous section: msText, msXLabel, msYLabel and msTitle. They all accept a character string as argument.

### 5.5.1 About the character sets used

The abovementionned character string contains usually ASCII encoded characters (ASCII set comprises 128 characters in all: most of them are glyphs, but there are also 32 non printable characters).

The string argument of the text routines supports 8-bits encoded characters, namely the ISO-8859-1 – or Latin-1 – character set (called for short "charset"). It adds to the 128 ASCII letters, other symbols and accentuated letters, mainly for european languages. By default, MUESLI suppose that the strings are in this charset.

MUESLI also accepts UTF-8 character string (some symbols are coded on two characters), but you must tell this to MUESLI by calling the msSetCharEncoding routine. However, only Latin-1 characters will be actually displayed.

▶ Example #5-15: (see output plot in figure 15)

```
call msFigure(1)
call msAxis( "off" )

call msText( 0.0d0, 0.6d0,                                              &
     "É È Ç À Ù Û Ü Î Ï Ô      é è ç à ù û ü î ï ô", height=2.0d0 )

call msText( 0.0d0, 0.4d0,                                              &
     "Â Ã Ä Å Æ Ñ Õ Ö Ø Ý     â ã ä å æ ñ õ ö ø ý", height=2.0d0 )
```

### 5.5.2 Greek alphabet and other symbols

Greek letters are available either via the short escaped sequence: \g*l*, where *l* is a latin letter, or via the explicit \(greek_letter_name) escaped sequence inside the character string, as in the table 1.

The Hershey symbols can be directly called by their symbol number, as follows: \(nnn), where nnn is the symbol number (any number of digits).

Commonly used symbols are summarized in the table 2.

40 standard graph markers (ranged from -8 to 31) are available via the \mn (or \mnn) escaped sequence, and are drawn on figure 16.

Lastly, concerning the layout of the characters, you may refer to the table 3 for producing superscript, subscript and superposition of characters (note that \u and \d should always be used in pairs) and also fine tuning for spaces between characters.

| **Greek letter** | **explicit escaped sequence** | **short escaped sequence** |
|---|---|---|
| $\alpha$ | `\(alpha)` | `\ga` |
| $\beta$ | `\(beta)` | `\gb` |
| $\gamma$ | `\(gamma)` | `\gg` |
| $\delta$ | `\(delta)` | `\gd` |
| $\epsilon$ | `\(epsilon)` | `\ge` |
| $\zeta$ | `\(zeta)` | `\gz` |
| $\eta$ | `\(eta)` | `\gh` |
| $\theta$ | `\(theta)` | `\gq` |
| $\iota$ | `\(iota)` | `\gi` |
| $\kappa$ | `\(kappa)` | `\gk` |
| $\lambda$ | `\(lambda)` | `\gl` |
| $\mu$ | `\(mu)` | `\gm` |
| $\nu$ | `\(nu)` | `\gn` |
| $\xi$ | `\(xi)` | `\gx` |
| $o$ | `\(omicron)` | `\go` |
| $\pi$ | `\(pi)` | `\gp` |
| $\rho$ | `\(rho)` | `\gr` |
| $\sigma$ | `\(sigma)` | `\gs` |
| $\tau$ | `\(tau)` | `\gt` |
| $\upsilon$ | `\(upsilon)` | `\gu` |
| $\phi$ | `\(phi)` | `\gf` |
| $\chi$ | `\(chi)` | `\gc` |
| $\psi$ | `\(psi)` | `\gy` |
| $\omega$ | `\(omega)` | `\gw` |
| $\vartheta$ | `\(vartheta)` | — |
| $\varpi$ | `\(varpi)` | `\gv` |
| $\varsigma$ | `\(varsigma)` | — |
| $\varphi$ | `\(varphi)` | `\gj` |

Table 1: Escaped sequences for greek letters – Uppercase glyphes are obtained in changing the first letter in capital (variants have the same capital letter as the basic ones).

| **Symbols** | **escaped sequence** |
|---|---|
| Å(Ångström) | `\A` |
| × (times) | `\x` |
| · (centered dot) | `\.` |
| \ (backslash) | `\\` |
| œ (ligature) | `\oe` |
| Œ (ligature) | `\OE` |
| $^1$ (1 as superscript) | `\1` |
| $^2$ (2 as superscript) | `\2` |
| $^3$ (3 as superscript) | `\3` |

Table 2: Escaped sequences for commonly used symbols (superscripts 1, 2 and 3 are only valid for the default font!)

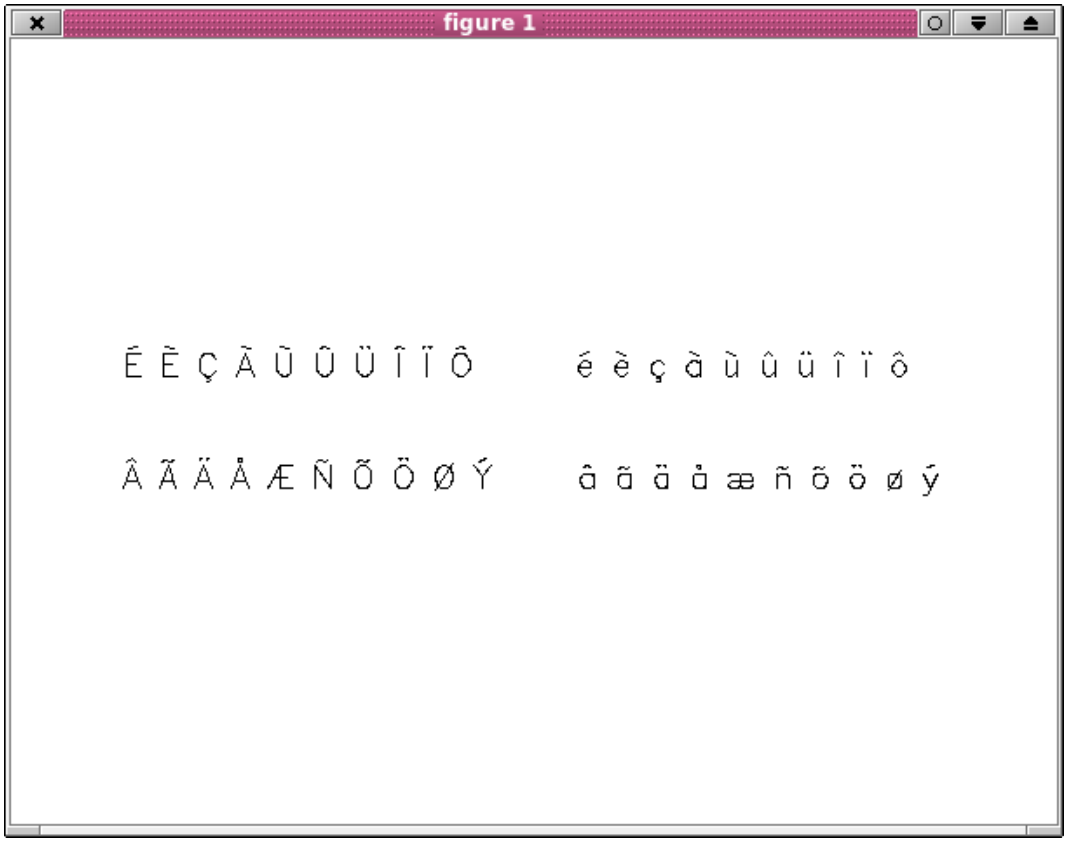



Figure 15: output of ex. #5-15: Latin-1 character string

### 5.5.3 Fonts available

There are four different fonts; they are:

- normal, abbreviated by the letter **n**; this is the default font. It is a sans-serif font (like *Helvetica*). Again, be aware that only this font supports all the accentuated letters defined in the ISO-8859-1 charset.
- roman, abbreviated by the letter **r**. It is a serif font, like *Times*.
- italic, abbreviated by the letter **i** (like *Times-Italic*).
- script, abbreviated by the letter **s**. It actually looks close to the *English* script font.

You are able to switch from one font to another one in the same character string, by using the escaped sequences: `\fn`, `\fr`, `\fi` and `\fs` (be aware that the interpreter of the escaped sequences is case sensitive).

| Layout | escaped sequence |
|---|---|
| up one level (superscript) | `\u` |
| down one level (subscript) | `\d` |
| backspace (superposition) | `\b` |
| standard space | `\s1` |
| thin space | `\s2` |
| very thin space | `\s3` |

Table 3: Layout escaped sequences

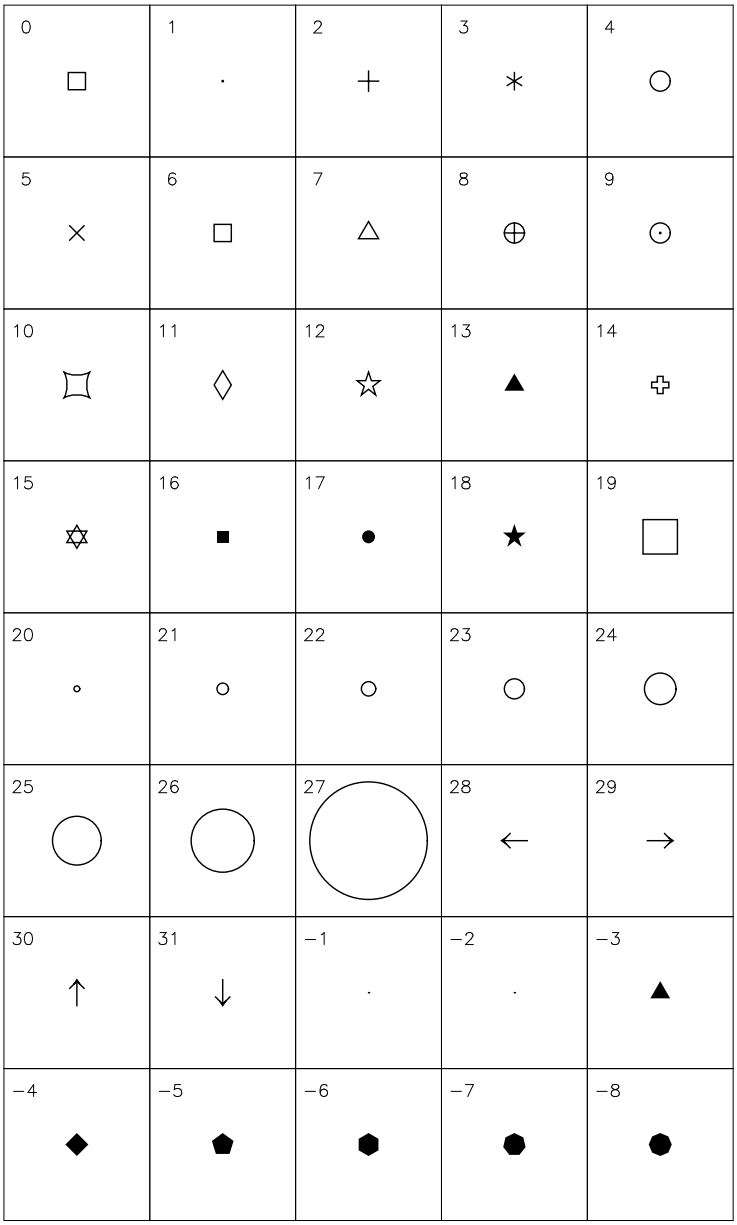



Figure 16: Standard graph markers

▶ Example #5-16: (see output plot in figure 17)

```
! real(kind=MF_DOUBLE) :: x = 0.0d0

call msFigure(1)
call msAxis( "off" )

call msText( x, 0.6d0, "\fn A B C D a b c d \fr A B C D a b c d",    &
             height=2.0d0 )
call msText( x, 0.4d0, "\fi A B C D a b c d \fs A B C D a b c d",    &
             height=2.0d0 )
```

Greek letters are exactly the same in all these fonts.

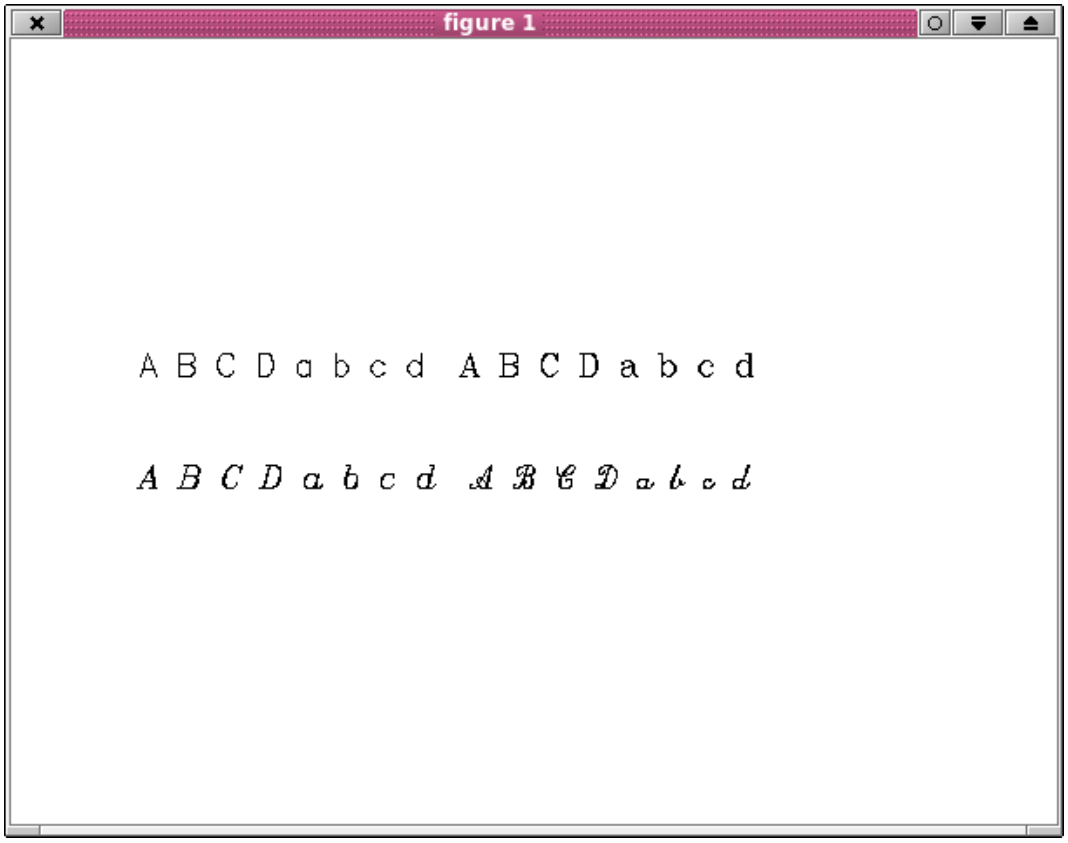



Figure 17: output of ex. #5-16: using different character fonts
(from left to right and top to bottom, normal font, roman font, italic font and script font).

## 5.6 Printing Figures

When you print the current figure (`msPrint` routine), all the graphic objects are drawn once again in another device. Only two devices (or formats) are available: EPS and PNG.

The EPS (Encapsulated PostScript) format corresponds to a vectorized description of the figure. So, it can be magnified at any level, without altering the quality of the graphics. Moreover, bitmap objects, like 'patch', 'pcolor' or 'image' are compressed with the zlib, providing small size EPS files. The creator of the EPS file (*i. e.* the name of the executable) is added in the meta info only if the routine `msInitFgl` has been called at the beginning of the graphic part. Therefore, the creator will appear in the title when you open the EPS with ghostview, for example (besides, some File Managers like Konqueror in KDE will show this information in a popup window – see figure 18). Many applications can convert these EPS files in another vectorized format, like SVG.

The PNG (Portable Network Graphic) format is a modern bitmap format. It is made by a juxtaposition of colored pixels, so you cannot enlarge it too much. However, it is well adapted to the web or other media and can easily be converted into other bitmap format.

By default, all graphic commands are drawn in an X11 window. If your system has no graphic possibilities (or if you intent to execute your program as a batch), you can de-activate the X11 device by use of the routine `msX11Device`. In such a case, graphic commands are registered in memory so you can nevertheless print them in files.

Caution: with the current version of FGL, expect some problem when printing surface plots in a PNG. Use EPS format instead. Indeed, for the moment, the PNG file are limited to 256 colors.

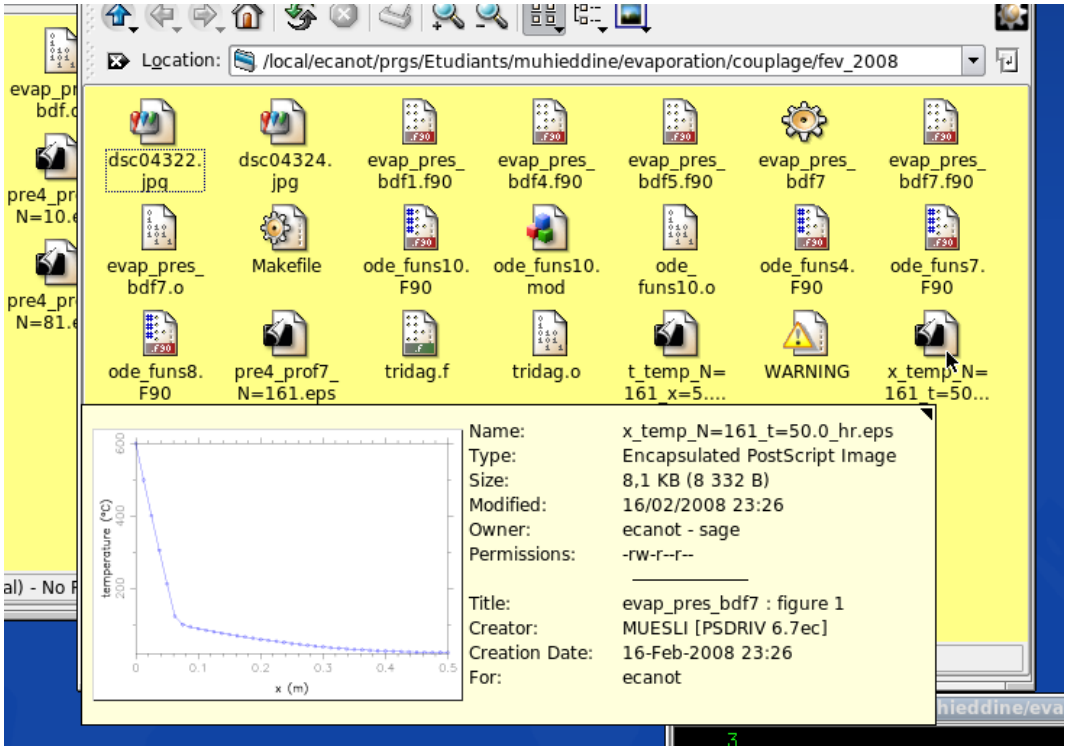



Figure 18: Meta Info showing the creator of the EPS file

## 5.7 Low-level Graphic Object's Manipulation

While it can be viewed as a high-level routine, the `msPatch` routine is used to draw one polygonal colored face. We prefer to describe it in this sub-section.

`msPatch`, like `msPColor`, accept an optional argument about transparency (actually, opacity) of the drawn object.

▶ Example #5-17: (see output plot in figure 19)

```
call msFigure(1)

call msAxis( [ 0.1d0, 0.8d0, 0.1d0, 0.8d0] )
call msAxis( "equal" )

call msColormapRange( 16, 255 )
call msColormap( "rainbow" )

x = [ 0., 1., 0. ] ! vertices x-coord.
y = [ 0., 0., 1. ] ! vertices y-coord.
c = [ 0., 1., 0. ] ! vertices color

call msPatch( x, y, c, transp=1.0d0 )

call msHold( "on" )

x = [ 1.1, 1.1, 0. ]
y = [  0., 1.1, 1.1 ]
c = [ 0.5, 0.5, 0.5 ]

call msPatch( x, y, c, transp=1.0d0 )

x = [ -0.1,  1.1,  1.1 ]
y = [ -0.1, -0.1,  1.1 ]
c = [  1., 0., 1. ]
call msPatch( x, y, c, transp=0.65d0 ) ! transparent object
call msDrawNow() ! tip for redrawing the framebox
```

`msGSet` is used to modify one property of the figure, or one property of a graphic object, which must be accessed by its handle. Therefore, if you plan to dynamically change a property by use of `msGSet`, you must draw the graphic object by the function version of the routine, in order to store its handle. Many properties can be changed, one at a time. See the *MUESLI Reference Manual* for a list of all possibilities.

`msRemoveDraw` is used to remove a graphic object. Be aware that the object is really deleted; if you want only make it hidden, you must use `msGSet` instead, and set the '`visible`' property to '`off`'.

`msGSet` and `msRemoveDraw` make their modifications only in the MUESLI graphic memory. To see the change effects on the screen, the user must call the `msDrawNow` which redraw all the objects of the current figure.

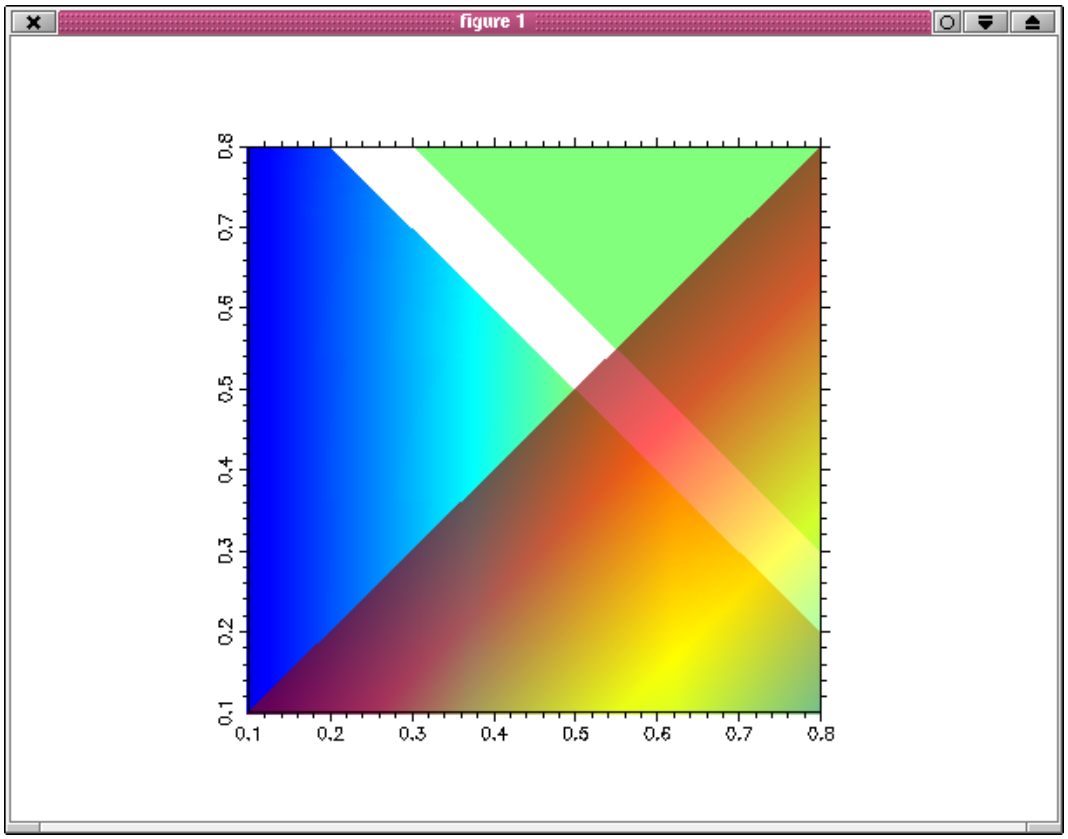



Figure 19: output of ex. #5-17: msPatch use

# Index

MUESLI file extensions


.mbf.gz ..... 38
.mbf ..... 38


MUESLI operators


** ..... 48
* ..... 21, 48
.and. ..... 23
.hc. ..... 19, 24, 38
.not. ..... 23
.or. ..... 23
.t. ..... 24
.vc. ..... 19, 24, 38
/= ..... 23
/ ..... 48
== ..... 23
> ..... 23


MUESLI parameters


MF_COLON ..... 14
MF_EMPTY ..... 14
MF_END ..... 14
MF_EPS ..... 62–64
MF_E ..... 13
MF_INF ..... 13
MF_I ..... 13
MF_NAN ..... 13
MF_PI ..... 13
MF_REALMAX ..... 62–64
MF_REALMIN ..... 62–64


MUESLI routines


All ..... 24, 25
Any ..... 24, 25
mf/msDaeSolve ..... 60
mfAll ..... 62–64
mfAny ..... 62–64
mfBand ..... 79
mfBesselI ..... 21
mfBesselJ ..... 21
mfBesselK ..... 21
mfBesselY ..... 21
mfCeil ..... 62–64
mfComplex ..... 62–64
mfCond ..... 28
mfConj ..... 62–64
mfCount ..... 62, 64
mfCshift ..... 62, 64
mfDaeSolve ..... 37, 56, 61
mfDiag ..... 24
mfEigs ..... 43
mfEoshift ..... 62, 64
mfErf ..... 21
mfExp ..... 21
mfEye ..... 10
mfFFT ..... 29
mfFSolve ..... 34, 56
mfFZero ..... 34
mfFigure ..... 65
mfFind ..... 29
mfFix ..... 62, 63
mfFlipLR ..... 24
mfFlipUD ..... 24
mfFloor ..... 62–64
mfFourierCos ..... 29
mfFourierLeg ..... 29
mfFourierSin ..... 29
mfFull ..... 42
mfGamma ..... 21
mfGetMsgLevl ..... 53
mfGetTrbLevel ..... 53
mfGet ..... 15
mfGinput ..... 79
mfGridData ..... 32, 33
mfImRead ..... 74
mfImag ..... 62–64
mfInterp2 ..... 32
mfInvFFT ..... 29
mfInvFourierCos ..... 29
mfInvFourierLeg ..... 29
mfInvFourierSin ..... 29
mfInv ..... 28
mfIsDense ..... 43
mfIsPosDef ..... 46
mfIsRowSorted ..... 46
mfIsSparse ..... 43
mfIsSymm ..... 46
mfLDiv ..... 26
mfLinSpace ..... 10
mfLoadAscii ..... 37
mfLoadHDF5 ..... 38
mfLoadSparse ..... 45
mfLoad ..... 45
mfLsqNonLin ..... 34–36, 56
mfMax ..... 29, 62–64
mfMean ..... 29
mfMerge ..... 62, 64
mfMeshGrid ..... 32
mfMin ..... 29, 62–64
mfMul ..... 22, 62–64
mfNnz ..... 63
mfNorm ..... 28
mfOdeSolve ..... 36, 37, 56
mfOnes ..... 10
mfPPVal ..... 31
mfPack ..... 62, 64
mfProd ..... 29, 62–64
mfQuad ..... 35
mfRDiv ..... 26, 27
mfRMS ..... 29
mfRand ..... 62–64

mfRank ..... 28
mfReadline ..... 15
mfReal ..... 62–64
mfRepMat ..... 20, 24, 62–64
mfReshape ..... 24, 62–64
mfRot90 ..... 24
mfRound ..... 62–64
mfSVDS ..... 43
mfShape ..... 62–64
mfSimpson ..... 35
mfSin ..... 21
mfSize ..... 62, 64
mfSort ..... 29
mfSpAlloc ..... 38
mfSpDiags ..... 38
mfSpEye ..... 38
mfSpImport ..... 40
mfSparse ..... 38
mfSpline ..... 31, 68
mfSqrt ..... 21
mfStd ..... 29
mfSum ..... 29, 62–64
mfSvd ..... 28
mfTrapz ..... 35
mfUnit ..... 48
mfUnpack ..... 62, 64
mfVar ..... 29
mfZeros ..... 10
mf_DE_Options ..... 56, 60
mf_NL_Options ..... 56
mf ..... 65
msAssign ..... 51
msChol ..... 28
msClf ..... 83
msContour ..... 68
msCustomMove ..... 79
msDisableFPE ..... 54
msDisplay ..... 11–13, 39, 47, 50
msDrawNow ..... 89
msEllipKE ..... 21
msEnableFPE ..... 54
msExitFgl ..... 65
msFigure ..... 65
msFind ..... 31
msFlops ..... 51, 52
msFreeArgs ..... 52
msFreePointer ..... 17
msFunFit ..... 34
msGSet ..... 89
msHist ..... 29
msHold ..... 66
msHorizConcat ..... 38
msImage ..... 74
msInitArgs ..... 52
msInitFgl ..... 65, 87
msLU ..... 26
msLegend ..... 81
msLoad ..... 38
msLoglog ..... 65
msLsqNonLin ..... 36
msMedian ..... 29
msMedit ..... 11
msMoveLegend ..... 79, 83
msOdeSolve ..... 37
msPColor ..... 68, 88
msPanAndZoom ..... 78
msPan ..... 78
msPatch ..... 88
msPlotCubicBezier ..... 66
msPlotCubicSpline ..... 67
msPlotHist ..... 75
msPlot ..... 65
msPointer ..... 17
msPolyFit ..... 34
msPrintHashes ..... 15
msPrintProgress ..... 15
msPrint ..... 87
msQR ..... 26
msQuad ..... 35
msReadHistoryFile ..... 15
msRelease ..... 52
msRemoveDraw ..... 89
msReturnArray ..... 52
msRowSort ..... 46
msSVD ..... 26
msSaveAscii ..... 37
msSaveHDF5 ..... 38
msSaveSparse ..... 45
msSave ..... 38, 45, 61
msSemilogx ..... 65
msSemilogy ..... 65
msSetAsParameter ..... 14
msSetAutoComplex ..... 55
msSetAutoRowSorted ..... 46
msSetBackgroundColor ..... 65
msSetCharEncoding ..... 83
msSetMsgLevel ..... 15, 51
msSetMsgLevl ..... 53
msSetPSFonts ..... 83
msSetRoundingMode ..... 54
msSetTrbLevel ..... 51, 53
msSetX11Device ..... 77
msSet ..... 15, 18, 42
msSpExport ..... 40
msSpReAlloc ..... 39
msSpy ..... 76
msSurf ..... 77
msSvd ..... 28
msText ..... 80, 83
msTitle ..... 80, 83
msUsePhysUnits ..... 47
msVertConcat ..... 39
msWriteHistoryFile ..... 15
msX11Device ..... 87
msXLabel ..... 80, 83
msYLabel ..... 80, 83
msZoom ..... 78